Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages

Shoaib Ashraf Kamil

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-1
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-1.html

January 2, 2013

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **02 JAN 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**
4. TITLE AND SUBTITLE **Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		

14. ABSTRACT

As the complexity of machines and architectures has increased, performance tuning has become more challenging, leading to the failure of general compilers to generate the best possible optimized code. Expert performance programmers can often hand-write code that outperforms compileroptimized low-level code by an order of magnitude. At the same time, the complexity of programs has also increased, with modern programs built on a variety of abstraction layers to manage complexity, yet these layers hinder efforts at optimization. In fact, it is common to lose one or two additional orders of magnitude in performance when going from a low-level language such as Fortran or C to a high-level language like Python, Ruby, or Matlab. General purpose compilers are limited by the inability of program analysis to determine programmer intent, as well as the lack of detailed performance models that always determine the best executable code for a given computation and architecture. The latter problem can be mitigated through auto-tuning, which generates many code variants for a particular problem and empirically determines which performs best on a given architecture. This thesis addresses the problem of how to write programs at a high level while obtaining the performance of code written by performance experts at the low level. To do so, we build domainspecific embedded languages that generate low-level parallel code from a high-level language, and then use auto-tuning to determine the best performing low-level code. Such DSELs avoid analysis by restricting the domain while ensuring programmers specify high-level intent, and by performing empirical auto-tuning instead of modeling machine parameters. As a result, programmers write in high-level languages with portions of their code using DSELs, yet obtain performance equivalent to the best hand-optimized low-level code, across many architectures. We present a methodology for building such auto-tuned DSELs, as well as a software infrastructure and example DSELs using the infrastructure, including a DSEL for structured grid computations and two DSELs for graph algorithms. The structured grid DSEL obtains over 80% of peak performance for a variety of benchmark kernels across different architectures, while the graph algorithm DSELs mitigate all performance loss due to using a high-level language. Overall, the methodology, infrastructure, and example DSELs point to a promising new direction for obtaining high performance while programming in a high-level language.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**183**	

Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages

By

Shoaib Ashraf Kamil

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Armando Fox, Co-Chair
Professor Katherine Yelick, Co-Chair
Professor James Demmel
Professor Berend Smit

Fall 2012

Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages

Abstract

Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages

by

Shoaib Ashraf Kamil

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Armando Fox, Co-Chair
Professor Katherine Yelick, Co-Chair

As the complexity of machines and architectures has increased, performance tuning has become more challenging, leading to the failure of general compilers to generate the best possible optimized code. Expert performance programmers can often hand-write code that outperforms compiler-optimized low-level code by an order of magnitude. At the same time, the complexity of programs has also increased, with modern programs built on a variety of abstraction layers to manage complexity, yet these layers hinder efforts at optimization. In fact, it is common to lose one or two additional orders of magnitude in performance when going from a low-level language such as Fortran or C to a high-level language like Python, Ruby, or Matlab.

General purpose compilers are limited by the inability of program analysis to determine programmer intent, as well as the lack of detailed performance models that always determine the best executable code for a given computation and architecture. The latter problem can be mitigated through *auto-tuning*, which generates many code variants for a particular problem and empirically determines which performs best on a given architecture.

This thesis addresses the problem of how to write programs at a high level while obtaining the performance of code written by performance experts at the low level. To do so, we build *domain-specific embedded languages* that generate low-level parallel code from a high-level language, and then use auto-tuning to determine the best performing low-level code. Such DSELs avoid analysis by restricting the domain while ensuring programmers specify high-level intent, and by performing empirical auto-tuning instead of modeling machine parameters. As a result, programmers write in high-level languages with portions of their code using DSELs, yet obtain performance equivalent to the best hand-optimized low-level code, across many architectures.

We present a methodology for building such auto-tuned DSELs, as well as a software infrastructure and example DSELs using the infrastructure, including a DSEL for structured grid computations and two DSELs for graph algorithms. The structured grid DSEL obtains over 80% of peak performance for a variety of benchmark kernels across different architectures, while the graph algorithm DSELs mitigate all performance loss due to using a high-level language. Overall, the methodology, infrastructure, and example DSELs point to a promising new direction for obtaining high performance while programming in a high-level language.

For all who made this possible,

and for June and Hass.

Contents

List of Figures

List of Tables

List of Symbols

ABI	Application Binary Interface
AMR	Adaptive Mesh Refinement
ANSI	American National Standards Institute
API	Application Programming Interface
Asp	Asp is SEJITS for Python
AST	Abstract Syntax Tree
ATLAS	Automatically Tuned Linear Algebra Software
BFS	Breadth-First Search
BLAS	Basic Linear Algebra Subprograms
BLB	Bag of Little Bootstraps
BSP	Bulk-Synchronous Parallel
BW	Bandwidth
CA	Communication Avoiding
CA-CG	Communication-Avoiding Conjugate Gradient
CG	Conjugate Gradient
CLI	Command Line Interface
CombBLAS	Combinatorial Basic Linear Algebra Subroutines
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
CUDA	Programming platform for Nvidia GPUs
DFS	Depth-First Search
DMA	Direct Memory Access
DP	Double Precision
DRAM	Dynamic Random Access Memory
DSEL	Domain-Specific Embedded Language
DSL	Domain-Specific Language
ECL	Embedded Common Lisp
EM	Expectation-Maximization
FFI	Foreign Function Interface
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
GB	Gigabyte
GCC	Gnu Compiler Collection
GEMM	General Matrix Multiply
GIL	Global Interpreter Lock

GPU	Graphics Processing Unit
HPC	High Performance Computing
HTTP	HyperText Transfer Protocol
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
JIT	Just-In-Time Compilation
JVM	Java Virtual Machine
KB	Kilobyte
KDT	Knowledge Discovery Toolbox
LAPACK	Linear Algebra Package
LLVM	Low-Level Virtual Machine
LOC	Lines of Code
MB	Megabyte
MKL	Intel Math Kernel Library
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
MSVC	Microsoft Visual C++ Compiler
MTEPS	Millions of Traversed Edges Per Second
NUMA	Non-Uniform Memory Access
ODE	Ordinary Differential Equation
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
OSKI	Optimized Sparse Kernel Interface
PDE	Partial Differential Equation
SAID	Semiring Additive Identity
ScaLAPACK	Scalable Linear Algebra Package
SDK	Software Development Kit
SEJITS	Selective Embedded Just-In-Time Specialization
SIMD	Single Instruction Multiple Data
SP	Single Precision
SPMD	Single Program Multiple Data
SpMV	Sparse Matrix-Vector Multiply
SQL	Structured Query Language
SSE	Streaming SIMD Extensions
SVM	Support Vector Machines
TDP	Thermal Design Power
TLB	Translation Lookaside Buffer
UMA	Uniform Memory Access
UPC	Universal Parallel C
VM	Virtual Machine

Acknowledgments

This thesis would not be possible without the guidance and support from my co-advisors, Professors Katherine Yelick and Armando Fox. Their rigorous questioning and enthusiasm for the work pushed me to solidify vague ideas into solid research, and their patience and ability to work with students inspires me. I would like to thank Professors James Demmel and Berend Smit for their valuable suggestions throughout the proposal, research, and writing process as well.

I must thank Professor James Demmel and Professor Yelick for encouraging me to pursue a career in computer science research. Since 2000, when I was a teaching assistant for Professor Demmel, I have almost continuously worked with them within the Berkeley Benchmarking and Optimization Group (BeBOP), at Lawrence Berkeley National Labs, and as a graduate student.

Thank you to Professors Armando Fox and David Patterson for teaching the original graduate seminar which spawned the idea and work embodied in this thesis, and their support for the ideas and push to make them the focus of my research.

Thank you to the BeBOP group in its various incarnations in the past 12 years, and especially Professor Rich Vuduc who turned my first attempts at research into publishable work. Extra thank yous to members of the group with whom I had insightful discussions and collaborations over the years: Rajesh Nishtala, Kaushik Datta, Ankit Jain, Marghoob Mohiyuddin, Mark Hoemmen, Oded Schwartz, and Benjamin Lipshitz.

I am indebted to Leonid Oliker and John Shalf for taking me under their wings during my tenure at Lawrence Berkeley National Laboratory, and teaching me how to conduct important research. I look forward to continuing our fruitful collaborations.

Without long discussions and input from Sam Williams, this thesis would not have been possible. Sam's efforts to quantify and explain every aspect of performance inspire much of the work in this thesis. Bryan Catanzaro's always helpful and critical input also heavily influenced me; it is almost unbelievable how we came to similar ideas and conclusions from very different starting points. Leo Meyerovich also helped solidify some of the ideas here in discussions during Par Lab retreats.

I would also like to thank the incomparable Par Lab Technical Support team of Jeff Anderson-Lee, Kostadin Ilov, and Jon Kuroda, who were always extremely helpful in responding to any esoteric requests I had about the many Par Lab machines and making sure the zoo of different architectures and form factors was running.

I am indebted to many of the Par Lab faculty who always made time to discuss research issues despite their already busy schedules, and in particular: Professors David Patterson, Ras Bodik, Koushik Sen, Krste Asanović, and Kurt Keutzer.

Thanks are due to the employees of Par Lab sponsors, especially from Intel and Microsoft, who were instrumental in the research presented in this thesis: Juan Vargas, Tim Mattson, Henry Gabb, and Adrian Chien.

The incredible administrative assistants for Par Lab and for Prof. Yelick at LBL deserve much thanks for always helping whenever I needed them: Roxana Infante, Tamille Johnson, Roxanne Clark, and Leah Temple.

Some of this work would be impossible without collaborating with the Knowledge Discovery Toolbox team: Prof. John Gilbert, Aydın Buluç, and Adam Lugowski. I must also thank Henry Cook and Ekaterina Gonina for their contributions and for being the first users of Asp.

Asp has benefitted from the work of an entire team of people, including Derrick Coetzee, Michael Driscoll, Jeffrey Morlan, and Richard Xia, along with a large number of undergraduate students.

Lastly but most importantly, I must thank my two families: first, the one I was born into, especially my parents, Mohammad and Shehnaz, who have accepted and become proud of my research career, and my siblings and their significant others, who kept me sane (or drove me insane, depending on the time). I must also thank my family of friends, especially those who were there beginning to end: Beto, Meg, Charlotte, Dipti, Samira, Solmaz, Sheila and Sheila, Suheir, Maisha, marcos, and Zara. Thank you, Dalia.

This work was performed at the UC Berkeley Parallel Computing Laboratory (Par Lab), supported by DARPA (contract #FA8750-10-1-0191) and by the Universal Parallel Computing Research Centers (UPCRC) awards from Microsoft Corp. (Award #024263) and Intel Corp. (Award #024894), with matching funds from the UC Discovery Grant (#DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract #DE-AC02-05CH11231.

Chapter 1

Introduction

A natural tension exists between the software goals of productivity and performance, with productivity enabled by abstraction layers that hide low-level hardware and software features, and performance enabled by implementing algorithms that carefully map onto those features. This thesis addresses the problem of how to write applications at a high level of abstraction while simultaneously obtaining performance that reflects highly-tuned code, sometimes near the hardware performance limit for a given computation. It addresses the ever-widening gap between performance and productivity, due to increases in complexity in both hardware and applications.

Performance tuning has become more challenging over time, and ultimately has led to the failure of general-purpose language compilers to generate highly-optimized code. One might see an order of magnitude better performance from hand-optimized code over compiled code, even for relatively low-level languages like C or Fortran. The end of clock speed scaling and the resulting multicore architectures and vector extensions have made this problem worse, since it adds automatic parallelization to the challenges of writing compilers.

At the same time, the complexity of programs has grown immeasurably over the past two decades. Whether the domain is web applications, computer games, or scientific simulations, the most complex of applications are built using a variety of abstraction layers—from system-level services and architecturally-tuned kernels at the bottom, to high-level languages and domain-specific application frameworks at the top. These abstraction levels are essential to managing software complexity, but add further challenges to performance optimization. It is not unusual to lose an additional order of magnitude in performance going from non-hand-optimized C or Fortran code, even when that code has not been highly tuned, to higher-level code in languages like Matlab, Python, Perl, or Ruby.

The effectiveness of general-purpose compilers is limited by two factors: the inability of program analysis techniques to determine what transformations are legal and the lack of accurate performance models to determine what code should be generated from among the legal possibilities. Even simple, easily-analyzable loop nests that operate on arrays, such as the three nested loops of matrix multiplication, can suffer from the latter. Automatic performance tuning or *auto-tuning* has emerged as a popular technique, taking a limited algorithmic domain and building a code generator that produces many possible implementations, typically in a language like C. The insight is that highly-optimized (but still portable) code can be written in C, if written carefully, so the most dramatic tuning can be done at the source level, producing an implementation that the compiler can effectively translate. The selection problem is handled by search—running each version and selecting the best-performing. The analysis problem is circumvented, because the code generator is written with the knowledge of what possible implementations are legal; for example, one does not transform three nested loops, but simply generates code that correctly implements matrix multiply. Libraries like FFTW [39], Atlas [118], SPIRAL [98], OSKI [115], and pOSKI [56] all use this idea, leveraging a wide range of techniques for producing the set of legal implementations and searching over them.

But while automatic tuning has produced high performance libraries, including libraries callable by scripting languages, there are several problems with this approach. First, code generators, often written using ad-hoc programming techniques, are themselves tedious and error-prone programs to write. Second, even a well-designed library interface can be cumbersome in the context of a high-level language. Libraries have particular problems when parameterized over functions, as in the case of structured grid computations or graph traversals that sweep over a data structure while applying some user-defined function. The traversals cannot be effectively tuned without including the operator; for example, optimizing a structured grid computation depends intimately on the number of neighboring values involved in each operator. Therefore, one cannot build a highly-optimized stencil library independent of the operators, and at the same time, the number of interesting operators is unbounded, so one cannot build a library instance for each one in advance.

This thesis introduces the concept of Selective Embedded Just-in-Time Specialization (SEJITS), which combines the power of high level scripting languages (including support for introspection and foreign function interfaces), domain-specific language extensions, dynamic code generation, and automatic performance tuning. The result is set of powerful domain-specific embedded languages (DSELs) that provide highly-optimized performance on a variety of machines, using software infrastructure we build to support this approach.

1.1 Thesis Contributions

This thesis makes the following contributions:

- We provide a framework in Chapter 5 for writing domain-specific code generators through a set of abstractions for manipulating and generating code and demonstrate their use in building auto-tuners.

- We demonstrate the use of introspection within a high level scripting language (Python) and use it to dynamically analyze and generate optimized versions for multiple architectures in our case studies (Chapters 7–13).

- We develop a technique for auto-tuning computations that involve higher-order functions, i.e., stencils or graphs, that can only be tuned after instantiation with a user-provided operator, in Chapters 10 and 12.

- We show, in Chapters 10 and 12, how an intermediate representation based on declarative semantics can provide the freedom needed to transform code without the need for difficult analyses.

- In Chapter 9, we demonstrate an approach for auto-tuning structured grid kernels using phased transformations combined with domain-specific knowledge. Unlike previous approaches that optimized single instances of a structured grid kernel, this proof-of-concept auto-tuning system can optimize many different kernels across many architectures.

- We demonstrate, in Chapter 10, a simple imperative language for expressing structured grid computations that is translated using introspection into a declarative intermediate form allowing for a large set of possible implementations. We demonstrate the high-level interface and its restrictions, which eliminate the need for complex analysis. The results show near-optimal performance.

- We show a second case study in Chapter 12 of graph traversal algorithms that uses an existing hand-tuned library (CombBLAS) and solves an important performance problem of optimizing over user-provided operators written in a high level language. The traditional approach of calling back to the high level language is prohibitively expensive. The resulting performance meets and sometimes exceeds hand-optimized code that is specialized for a particular operator.

- We demonstrate the effectiveness of our framework as a vehicle for delivering library auto-tuning to high-level languages for computations for which a full DSEL is unnecessary, in Chapter 13.

- In Chapter 13, we show that our infrastructure enables building DSELs that execute on GPUs, multicore CPUs, clusters using MPI, and in the cloud.

- We demonstrate the use of our framework by others in Chapter 13 who are not primary developers of the framework, showing that is a viable approach for performance programmers to use.

1.2 Thesis Outline

The following is an outline of the this thesis:

Chapter 2 provides background and motivates the need for high performance DSELs in high-level languages given current trends in hardware and software, including auto-tuning and auto-tuning compilers. We describe the Productivity-Performance gap, the central motivation for building high performance, high productivity systems.

In Chapter 3 we outline related work in the areas covered in this thesis, including DSELs, auto-tuning, and prior work for optimizing the two major case studies in the thesis: structured grid computations and graph algorithms.

Chapter 4 describes one of the central contributions of this thesis: the Selective Embedded Just-In-Time Specialization (SEJITS) methodology. This chapter explores the high-level ideas of the methodology and discusses the impact on delivering high performance and high productivity, even if only a subset of the ideas are used.

Chapter 5 introduces the Asp (Asp is SEJITS for Python) software infrastructure for implementing the SEJITS methodology in Python. It begins by demonstrating how a performance programmer can use the infrastructure to build small high-performance DSELs using Asp, then covers the different capabilities of Asp in more detail.

In Chapter 6 we describe the machine architectures used in this study as well as the methodologies for measuring performance, both as an absolute measure and as a fraction of theoretical peak performance for a given architecture.

Chapter 7 presents an overview of the six case studies in the thesis and outlines the importance of each as well as their computational domains.

Chapter 8 is an overview of the structured grid motif. This chapter illustrates the kinds of computations that make up the structured grid domain, how they differ from one another, and describes optimizations from the literature. Although this chapter is not meant to be comprehensive, it outlines some major aspects of this class of computation and gives some background on how they can be optimized.

Chapter 9 describes our first attempt at optimizing structured grid kernels in a general way using auto-tuning. Prior work has focused on individual applications, but this chapter shows that a framework based on code transformation can optimize a subset of structured grids using domain knowledge and obtain high performance across architectures, and across CPUs and GPUs. Performance is compared against theoretical peak, and results show the auto-tuner can obtain up to 98% of peak performance. We describe some limitations of this approach that motivate using DSELs instead of simple program transformations or external DSLs.

Chapter 10 is the first case study in this thesis, an auto-tuned high performance DSEL for structured grid computations. Taking the successes and lessons from the previous chapter into account, this DSEL optimizes a large class of structured grid computations while allowing users to express their computation in high-level Python code. Correctness is ensured by defining an intermediate representation that expresses only correct, compilable computations. Performance results show three orders of magnitude speedup versus pure Python and $100\times$ speedup over optimizing compilers. The DSEL outperforms state-of-the-art external DSLs [105] for structured grid computations while being far smaller and simpler, due to being implemented with Asp in a high-level language.

In Chapter 11, we introduce the graph algorithms motif and describe graphs, their properties, and common representations. Different applications, basic computations, and programming models are compared, concentrating on the linear algebra representation. The Knowledge Discovery Toolbox (KDT) [74], the Python package we use as a basis for our work, is described.

Chapter 12 demonstrates the SEJITS approach by building two different DSELs for graph analysis in KDT. The first enables users to restrict algorithm application to a subset of the graph using on-the-fly filtering, and improves performance to eliminate almost all overhead due to Python filtering. The second DSEL allows advanced users to write new graph algorithms in Python by defining the building blocks of KDT graph algorithms in high-level code. By compiling the DSEL into low-level code, over $2\times$ performance increases are obtained. This results in increased productivity for creating new graph algorithms in KDT.

Chapter 13 outlines three libraries built by others using Asp. First, we describe an auto-tuner for the matrix powers computation that occurs as a building block of communication-avoiding Krylov subspace methods [81] for solving linear equations. The resulting solvers can be written in Python yet outperform even highly-optimized vendor libraries. The second library implements the Expectation-Maximization algorithm for Gaussian mixture modeling [82] and uses auto-tuning to obtain high performance across platforms and even between CPUs and GPUs. With code generation from templated source snippets, this library outperforms the original hand-optimized implementations and is now the basis for research in the speech domain. Finally, we describe a

DSEL for the Bag of Little Bootstraps (BLB) [64] statistical method, showing that DSELs can use multi-backend code generation to run in the cloud as well as locally in parallel.

Chapter 14 synthesizes the results from our case studies and outlines lessons learned. We discuss insights from the implementations. Future work in high performance high productivity programming are described as well, both in the context of the SEJITS methodology as well as other complementary approaches. The thesis concludes with a summary of the many contributions and their potential impact.

Chapter 2

Motivation and Background

This chapter describes the necessary background motivating the need for a methodology and infrastructure to enable high performance and high productivity. Section 2.1 discusses current hardware trends, including the move to multicore. In Section 2.2, we describe trends for modern software development, especially in the arena of web applications. Section 2.3 outlines the major problem this thesis works to mitigate: the Productivity-Performance Gap. Section 2.4 outlines auto-tuning, one of the major methodologies we will leverage in this thesis. Finally, Section 2.5 summarizes.

2.1 Trends in Computing Hardware

For many decades of computer processor design, as new processors were built they followed Moore's Law, which predicts that the number of on-chip transistors doubles approximately every two years [83]. Much of this doubling is due to advances in complementary metal-oxide-semiconductor (CMOS) technology, which results in smaller and smaller feature sizes in integrated circuits. As a result, the feature sizes have shrunk from 500 nanometers to 22 nanometers in the span of 22 years.

Previously, each shrink allowed a commensurate increase in CPU clock rate. However, due to power and cooling limits at lower feature sizes, clock rates have reached a plateau and can no longer be scaled higher; since around 2004, mainstream processors have hovered around the same clock speeds. Figure 2.1 shows the scaling of the number of transistors as well as clock speeds for historical processors, illustrating the clock speed plateau.

Instead, processor manufacturers made a gamble on multicore: using the increased number of transistors to build many identical processors on the same die, connected by an integrated on-chip interconnect. Unlike scaling processor speeds, there is no automatic performance increase from multiple cores. In order to fully utilize multiple processors, code must be reprogrammed with parallelism in mind. As Figure 2.1 shows, the increase in the number of cores began in earnest as clock scaling stopped, continuing the applicability of Moore's Law.

As the number of cores in each CPU increases, there is a need to increase memory bandwidth between DRAM and CPU commensurately in order to obtain the same performance. Thus, there is a trend to increase the number of channels between the two, though such a trend cannot continue indefinitely. Furthermore, on today's processors, fully saturating memory bandwidth requires using more than a single core. Thus, parallelism is necessary even for applications that are bound by

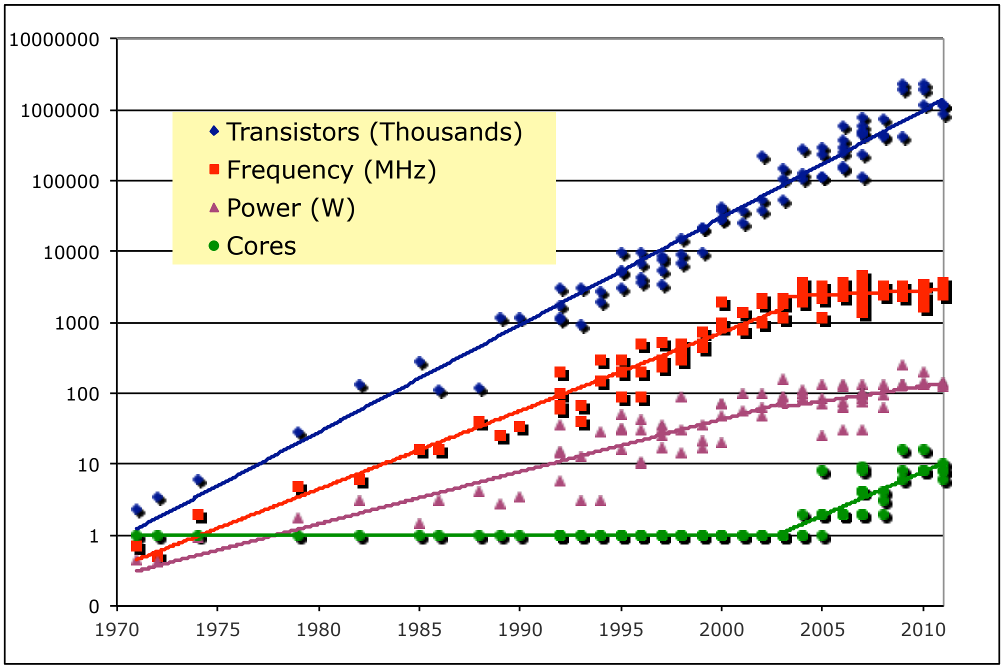


Figure 2.1: Historical processor trends, 1975-2011. This graph demonstrates that although Moore's Law continues, resulting in increases in transistor counts, clock speed increases have stalled. Additional transistors are being used by increasing numbers of cores in processors. Data gathered by Herb Sutter, Kunle Olokotun, Lance Hammond, Burton Smith, Chris Batten, Krste Asanović, Angelina Lee, and Katherine Yelick.

memory bandwidth performance, not just those bound by CPU performance.

2.2 Trends in Software

High-level languages have found footing in domains such as web programming, but lack support for parallelism or performance programming. Another term for many of these language, "scripting languages," refers to the class of programming languages whose primary purpose is to enable calling other programs and composing them, while performing intermediate tasks such as parsing one's output to alter and pass to another program. Examples of such scripting languages include Tcl [107], Perl [108], Python [100], and Ruby [102]. Such languages contain relatively high-level constructs as well as feature-rich standard libraries and an ecosystem of third-party addons that are easy to install. They have found widespread use among system administrators and users for performing common tasks. Performance is not a first-class concern for such languages, but these tasks usually do not require it.

Recently, with the emergence of the internet as a first-class platform, such scripting languages have been widely used to build web applications due to their high-level nature, familiarity, and ease-of-use. One of the major languages to emerge for this field is Ruby, which, along with the web programming package Ruby on Rails, has enabled rapid development of web applications. Because Ruby has strong metaprogramming support, the Ruby on Rails framework extensively uses metaprogramming to increase productivity, and, as a result, many new programmers are becoming familiar with high-level languages and their potential. In fact, some case studies have demonstrated 2—5× increases in productivity due to using high level languages rather than low-level ones [24, 53, 96].

In addition, Ruby on Rails and similar frameworks make extensive use of domain-specific languages (DSLs) for tasks such as database creation and expressing relationships between data, provisioning servers, and testing. These DSLs may be as simple as thin veneers over application programming interfaces (APIs) or as complex as declarative languages that are dynamically interpreted or compiled. For example, the ActiveRecord package [48] in Ruby on Rails allows programmers to express database queries as successive method calls on objects; these calls are dynamically translated into optimized structured query language (SQL) for use with backend databases. Such a DSL that uses the host language's syntax yet alters the semantics is called an *domain-specific embedded language* (DSEL). Thus, programmers are becoming more familiar with programming using many different DSLs, each suited for a particular type of task.

Due to the multicore revolution, parallel programming is becoming more important for obtaining sufficient performance. However, these high-level languages generally have poor support for parallel programming. Many scripting language interpreters employ non-threadsafe internal structures that must be protected from concurrent access. For example, in the standard Python interpreter, a Global Interpreter Lock (GIL) prevents more than one thread from running at once in order to protect the interpreter's state [9]. Furthermore, these languages are almost always garbage collected, but the collectors are rarely threadsafe. Hence, even if parallelism support is added to scripting languages, rewriting large portions of the interpreters will be necessary.

To obtain high performance, programming current multicore processors requires intense low-level tuning, paying careful attention to details of both the operations required by the application and intricacies of the processor, memory system, and available parallelism. Because of the inherent overheads of interpreting a language and due to the inability of these languages to allow users to express low-level data movement and computation, it is unlikely that simply adding parallelism would bridge the 3 orders of magnitude performance differences between high-level and low-level languages.

2.3 The Productivity-Performance Gap

Domain scientists or *domain experts* are scientists who utilize computers in order to perform scientific investigations, such as simulating the climate or analyzing structures of social networks. Such scientists lack a strong programming background or formal computer science training, and instead prefer to think of their problems in a way that maps to their domain. In particular, such scientists prefer to program in Matlab or some other high-level language that consists of operations they are familiar with, even though such programs may not meet necessary performance goals when running on realistic data sizes. Since productivity is their main goal, we refer to such programmers *productivity programmers*, and the languages they use as productivity-level languages (PLLs).

Indeed, one common pattern for such science is for domain scientists to prototype their algorithms in Matlab using small datasets until they are reasonably confident of correctness. Then, the program is passed to experts in programming efficient, low-level code, whom we call *efficiency programmers*. These programmers focus less on the underlying science behind the application, but instead concentrate on rewriting correct prototypes into low-level parallel implementations that can achieve orders of magnitude faster performance. Such performance is often necessary for the scientific results to have meaning; for example, climate simulation must be run with quite large datasets in order to obtain reasonable results for large timescales. If the program is not optimized, the performance can be too slow to give usable results [117].

This two-phase approach results in a number of problems. First, the efficiency programmers must re-implement already-correct code, which is an opportunity for introducing errors due to mistranslation or not understanding all details of the original implementation. In addition, this faster version intersperses machine-specific and performance-specific code within the algorithm. The result is a program where algorithmic and performance considerations are mixed together, and can be difficult to maintain given that the efficiency programmer is generally not proficient in algorithmic aspects while the productivity programmer is not familiar with performance-conscious constructs. This performance is rarely cross-platform, since even slight revisions to architectures change how optimal code should be structured. Finally, the approach is not scalable: it requires many programmer-hours to rewrite code for each platform in order to obtain good performance.

This implementation anti-pattern [67] is the result of a tradeoff programmers must make: they can either program in high level languages and obtain $5\times$ the productivity, or program in low-level efficient languages and obtain $5--10\times$ the performance. We call this the *Productivity-Performance Gap* [22]. This thesis explores how to bridge this gap and eliminate the tradeoff.

2.4 Auto-tuning and Auto-tuning Compilers

Although modern optimizing compilers incorporate complicated analyses and can perform auto-parallelization, auto-vectorization, and a host of other optimization techniques, they cannot always generate the best compiled code for the programmer's computation. This is due to the fact that general compilers must infer programmer intent from low-level code, and therefore can only use program text and analysis to determine whether optimizations are safe. Compilers must be conservative in their application of optimizations.

As a result, programmers who are in need of the best possible performance perform domain-specific optimizations by hand, either by altering the source code or by hand-writing assembly for the most performance-critical parts of the application. Because language-level constructs in low-level languages have semantics that convey little information about the higher-level computation, performance programmers can often hand-rewrite and hand-transform code in ways that may be unsafe if applied generally but safe given the computation at hand. For example, in C, arrays are encapsulated by a pointer to the beginning and indexing uses pointer arithmetic. For a code that reads from one array and writes to another, the compiler may not be able to prove that the pointers point to strictly disjoint pieces of memory, and may therefore not be able to reorder loop iterations (this problem is referred to as pointer aliasing). However, a programmer who knows that the application is always used in a manner that prevents overlapping arrays can hand-apply these optimizations.

However, hand-tuning is tedious and fragile, since programmers must write many more lines of code to implement these optimizations. Furthermore, the hand-tuning usually is highly machine-specific; for example, when rewriting performance-critical sections to use SIMD instructions, the specific instructions available differ from platform to platform and even from generation to generation from the same manufacturer.

An alternative to manual tuning is *auto-tuning*, which uses code generation to generate many different versions of the same code with different code optimizations applied, then runs all of them to determine empirically which version is appropriate for the set of inputs on that particular machine. For example, an auto-tuner for matrix multiply may apply different loop blockings for each version. Auto-tuning was first applied to this domain in the PhiPAC research project [14]. In addition to generating many versions, each version uses a subset of the low-level language that is easy for compilers to analyze and does not require extensive compiler transformations. This approach essentially uses the low-level language (C, C++ , or Fortran) as the assembly language. Auto-tuning enables portable performance as long as the code is auto-tuned on each platform separately.

Auto-tuned libraries are limited in that they are only suitable if the computation at hand can be expressed in a library, partially due to high performance codes being written in languages that do not have support of higher-level programming features. Not all computational motifs can be encapsulated in a library: for example, in structured grid computations, the function applied at each point changes from application to application, making it difficult to turn into a library in low-level languages that do not support higher-order functions. However, the approach has demonstrated successfully that domain-specific optimizations can be encapsulated and use code generation for cross-platform high performance.

Besides libraries, it is possible to implement auto-tuning within a general compiler; instead of using hard-coded heuristics to generate assembly that implements a function, the compiler can empirically run different versions. However, this approach is still hampered by the lack of high-level domain-specific information at the level the compiler is operating, since it only can manipulate the low-level code and must determine what optimizations to apply through analysis of this low-level version.

2.5 Summary

While trends in hardware have resulted parallelism being required to obtain performance gains, programmers are increasingly using higher-level languages that allow concise expression of computation. However, obtaining performance requires manipulating data and computation at low levels with deep knowledge of the particular hardware. Auto-tuning is one means for automatically determining the best way to write a particular library operation, without needing to understand all of the complexity of modern hardware. This thesis utilizes work in compilers, auto-tuning, and domain-specific languages to build a methodology for enabling high performance highly productive programming, a methodology that is necessary if future non-expert programmers are to benefit from the multicore revolution while still programming in productive, high-level languages.

Chapter 3

Related Work

This thesis spans several areas of research, combining performance optimization, domain-specific languages, and acceleration of particular types of computation. In this chapter, we outline related work from the various areas. In Section 3.1 we describe related work for improving performance of computational kernels. Section 3.2 outlines approaches for increasing the performance of Python, the host language for our domain-specific embedded languages. Related work in DSELs is described in Section 3.3. Just-in-Time compilation and specialization are covered in Section 3.4. Section 3.5 describes related work in optimizing structured grid algorithms, and Section 3.6 covers some related work in improving graph algorithm performance. Finally, Section 3.7 summarizes the related work.

3.1 Optimized Low-level Libraries and Auto-tuning

Auto-tuning was first applied to dense matrix computations in the PHiPAC library (Portable High Performance ANSI C) [14]. Using parameterized code generation scripts written in C, PHiPAC generated variants of general matrix multiply (GEMM) with a number of optimizations plus a search engine, to, at install time, determine the best GEMM routine for the particular machine. The technology has since been broadly disseminated in the ATLAS package [118]. Auto-tuning libraries include OSKI [115] and pOSKI [56] (sparse matrix-vector multiplication), FFTW [39] (Fast Fourier Transforms), SPIRAL [98] (signal processing transforms), and stencils (Pochoir [105] and others [30, 59]), in each case showing large performance improvements over non-auto-tuned implementations. With the exception of SPIRAL and Pochoir, all of these code generators use ad-hoc Perl or C with simple string replacement, unlike our template and tree manipulation systems. Efforts to integrate auto-tuning with compilers [112] have resulted in speedups for certain applications and kernels, but are not fully-automatic, requiring programmer intervention to guide the compiler.

The OSKI (Optimized Sparse Kernel Interface) sparse linear algebra library [115] precompiles 144 variants of each supported operation based on install-time hardware benchmarks that take hours to run, and includes logic to select the best variant at runtime, but applications using OSKI must still intermingle tuning code (hinting, data structure preparation, etc.) with the code that performs the calls to do the actual computations. The parallel version of OSKI, called pOSKI [56], explodes the search space due to including parallel parameters as part of the auto-tuning.

Petabricks [5] is a language and compiler that treats algorithmic choice as a first-class part of the language. Users can specify many different ways to compute the same thing, and the compilation system uses auto-tuning to generate best implementations. Active Harmony [26] is a similar system for full applications that tunes performance for applications with many components, each with many degrees of freedom. An alternative approach to optimized low-level code is to use synthesis, such as that provided by the Sketch [103] compiler. In this case, the compiler uses as input optimized skeleton code with holes along with a non-optimized specification and generates optimized code using the skeleton that executes (provably) the same computation as the non-optimized code for all inputs.

Inspector-executor strategies [79] apply automatic tuning to irregular applications by examining data structures at runtime to determine the best strategy for execution on a parallel machine. Automatic frameworks for such computations are currently under development [68], and show the potential for run-time auto-tuning for irregular computations on current architectures.

Most widely-used libraries written in efficiency languages do not gracefully handle higher-order functions as required for the structured grid and graph DSELs—even if the productivity language from which they are called does support them. This is usually because the efficiency languages do not have well-integrated support for higher-order functions themselves. Even libraries such as Intel's Array Building Blocks [85] or others using C++ expression templates cannot benefit from all the runtime knowledge available to DSEL compilers.

3.2 Accelerating Python

A popular way to provide good performance to productivity-language programmers has been to provide native libraries with high-level language bindings such as SciPy [57] and NumPy [88] (which collectively provide almost equivalent functionality to Matlab for Python programmers) and Biopython [27]. However, this approach can only be applied to domains where libraries are the appropriate delivery vehicle, and does not apply to user-written code.

The Weave subpackage of SciPy allows users to embed C++ code in strings inside Python code; the C++ code is then compiled and run, using the NumPy and Python C APIs to access Python data. Cython [11] is an effort to write a compiler for a subset of Python, while also allowing users to write extension code in C. Similarly, the ctypes package is a part of the standard library in Python that allows users to interface with C libraries by providing information about types and library functions, enabling the wrapping of C libraries to move from C code to Python code.

Closer to our own approach is Copperhead [22], which provides a Python-embedded DSEL that translates data-parallel operations into CUDA GPU code. Compilation is triggered by the presence of annotations, unlike our class-based approach. Copperhead utilizes PyCUDA [66], a library similar to CodePy [65], which we use. These enable expressing CUDA or C++ code as mixed trees and strings in Python; the libraries can compile the expressed low-level code and execute it, returning results to the user.

3.3 Domain-Specific Embedded Languages

Domain-specific embedded languages are used in many programming languages such as Lisp (with its macro system), Scheme and Racket [93] (which allows even changing the syntax for DSELs) and Haskell [71]. The term DSEL is due to Hudak, who describes circumstances in which such languages are useful [52]. Similarly, Mernik [78] attempts to derive a pattern language for domain-specific languages, included those that are embedded in a host language.

Like the Delite project [23] (the most similar recent work to our own), we exploit domain-specific information available in a DSEL to improve compilation effectiveness. In contrast to that work, we allow compilation to external code directly from the structural and computational patterns expressed by our DSELs; in the Delite approach, DSELs are expressed in terms of lower-level patterns and it is those patterns that are then composed and code-generated for. Put another way, by eschewing the need for intermediate constructs, SEJITS "stovepipes" each DSEL all the way down to the hardware without sacrificing either domain knowledge or potential optimizations. The most prominent example of this is that SEJITS allows using efficiency-language templates to implement runtime-generated libraries (or "trivial" DSLs). Furthermore, auto-tuning is a central aspect of our approach, enabling high performance without needing complex machine and computation models.

For C++ , the Proto toolkit [86] simplifies the creation of DSELs that use expression templates. Like Asp, Proto advocates the use of grammar-based DSELs and makes their implementation far simpler; however, Proto does not enable run-time specialization of the code, sticking to the C++ compile-time model.

3.4 Just-in-Time Compilation & Specialization

Early work on specialization appeared in the Synthesis Kernel, in which a code synthesizer specialized kernel routines on-the-fly when possible [97]. Engler and Proebsting [36] illustrated the benefits of dynamically generating small amounts of performance-critical code at runtime. Jones [58, 42] and Thibault and Consel [111] proposed a number of runtime specialization transformations to increase performance of programs, including partial evaluation or interpreters customized for specific programs. Despite their different contexts, these strategies, like SEJITS, rely on selectively changing execution of programs using runtime as well as compile-time knowledge.

Sun's HotSpot JVM [91] performs runtime profiling to decide which functions are worth the overhead of JIT-ing, but must still be able to run arbitrary Java bytecode, whereas SEJITS does not need to be able to specialize arbitrary productivity-language code. Our approach is more in the spirit of Accelerator [106], which focuses on optimizing specific parallel kernels for GPUs while paying careful attention to the efficient composition of those kernels to maximize use of scarce resources such as GPU fast memory. Asp is more general in allowing DSEL implementers to use the full set of Python features to implement DSEL compilers that capture their problem-solving strategy.

PyPy [16] is a reimplementation of the Python interpreter that uses translation from Python to a restricted typed subset (called RPython) along with a Just-in-Time interpreter. Unlike the JVM and Microsoft's CLI [104], the virtual machine for RPython is designed from the ground up for dynamic languages. This approach yields a $5.5\times$ speedup over the standard Python interpreter, but the current PyPy implementation does not support many of the libraries needed by users in scientific programming, mainly due to insufficient support in PyPy for the C API used by extensions for the standard Python interpreter. Nevertheless, PyPy's translation machinery is a potential interface for creating new domain-specific translations that run (perhaps only partially) on the PyPy VM.

3.5 Accelerating Structured Grid Computations

Cache optimizations for structured grid computations are the subject of much prior work. Datta et al [31] evaluated a number of optimizations for serial structured grid calculations, including the partial 2D blocking of 3D grids due to Rivera and Tseng [101], and time skewing [122], which blocks in both space and time to optimize cache reuse, as well as the cache-oblivious stencil algorithm [40], which successively recurses on subsets of the structured grid in order to perform the calculation on a working set that fits in cache and is implicitly blocked in time and space.

Auto-tuning structured grid calculations for parallel multicore using code generation scripts is the subject of Datta's thesis [30] and a large set of possible optimizations are described in Williams's thesis [121]. The latter applies auto-tuning to LBMHD [120], a lattice Boltzmann application with several complicated structured grid kernels. In both of these cases, the auto-tuning is specific to the application or kernel. However, the set of optimizations forms a basis for building a more general structured grid optimization framework.

PATUS [25] is a Java reimplementation of the structured grid auto-tuner in Chapter 9, with similar functionality as the original, utilizing the same set of optimizations. Unlike our auto-tuner, it allows users to define their own optimization strategies. Pochoir [105] is a domain-specific language compiler that performs parallel cache-oblivious optimization of structured grid calculations. Unlike our work, there is no auto-tuning utilized, and the DSL is not embedded or high-level.

A rich area of research into general optimizing compilers that has high applicability to structured grid computations is the polyhedral model [45], which is an analysis applied to nested loops (such as those that occur in structured grid computations) that treats them as a traversal of points bounded by a polyhedron. The analysis attempts to reorder the computation (adding parallelism or reducing cache traffic) while preserving dependencies. It has been applied to structured grid algorithms [17] with some success. Unlike our approach, it does not use domain-specific knowledge or auto-tuning, instead relying on general compiler analysis and heuristics. Polyhedral loop optimization is integrated into the Gnu Compiler Collection (GCC) via the Graphite [94] package.

3.6 Accelerating Graph Algorithms

The Graph500 Benchmark [28] is the modern benchmark for measuring graph algorithm performance. The major phase in the algorithm uses a set of Breadth First Searches (BFSs) and measures the performance in terms of millions of traversed edges per second (MTEPS) and ranks the top 500 machines, similar to the Top 500 HPC benchmark [113].

A number of recent packages have been built to enable high speed graph analysis. Pregel [75] was Google's infrastructure used for certain graph algorithms performed on their large database of web pages, though it has been superceded for internal use. The Combinatorial BLAS [20] is a set of low-level algorithms that treat graphs as matrices and graph operations as linear algebra with specialized semirings, allowing the package to implement well-known optimizations from the linear algrebra domain to obtain excellent scaling and performance.

3.7 Summary

In this thesis, drawing on the existing research in auto-tuning and DSELs, we apply just-in-time code generation and compilation techniques to DSELs in Python. We use run-time auto-tuning in order to obtain best performance. Our major case studies, which describe DSELs for structured grid computations and graph algorithms, draw on and advance the state of the art in both areas, combining new techniques with previous research; in some cases, we extend the applicability of previous work with our easier-to-use system for packaging domain-specific expertise in a way usable for non-expert programmers.

Chapter 4

SEJITS: A Methodology for High Performance Domain-Specific Embedded Languages

In this chapter, we outline the Selective Embedded Just-In-Time Specialization (SEJITS) methodology for domain-specific embedded languages. SEJITS enables small, easy-to-build DSEL compilers for high-level languages, enabling high performance and parallelism. With SEJITS, programmers who are not experts in performance programming can program in high-level languages while enjoying the performance benefits of low-level parallel languages and the knowledge of performance experts.

We begin with an overview of the SEJITS methodology in Section 4.1. Section 4.2 outlines DSELs in productivity languages and contrasts them with application programming interfaces (APIs). Sections 4.3 and 4.4 talk about two mechanisms essential to SEJITS: code generation and auto-tuning. Section 4.5 describes some best practices for building DSELs in the methodology. In Section 4.6 we outline requirements for hosting languages for the SEJITS methodology. Section 4.7 summarizes the chapter.

4.1 Overview of SEJITS

In this chapter, we will use a simple example to motivate the methodology. Consider a *domain scientist* who is an expert in a particular scientific area, but is not an expert programmer. The domain scientist wishes to write a scientific application that simulates a physical process using a new method, and within this application, several computational kernels are used, including a graph algorithm kernel (see Chapter 11). However, if these kernels do not run with high performance, the prototype application will not be able to yield useful results. The SEJITS methodology aims to enable this high performance.

An overview of the SEJITS methodology is shown in Figure 4.1. In the methodology, domain-specific portions of high-level *productivity layer* user code (written in a language such as Matlab, Python, or Ruby) is transformed into high-performance candidates in low-level *efficiency layer* code (in a language such as C, C++ , or CUDA) automatically, and run on the low-level parallel hardware, returning results back to the high-level interpreter. Subsequent runs utilize different generated code

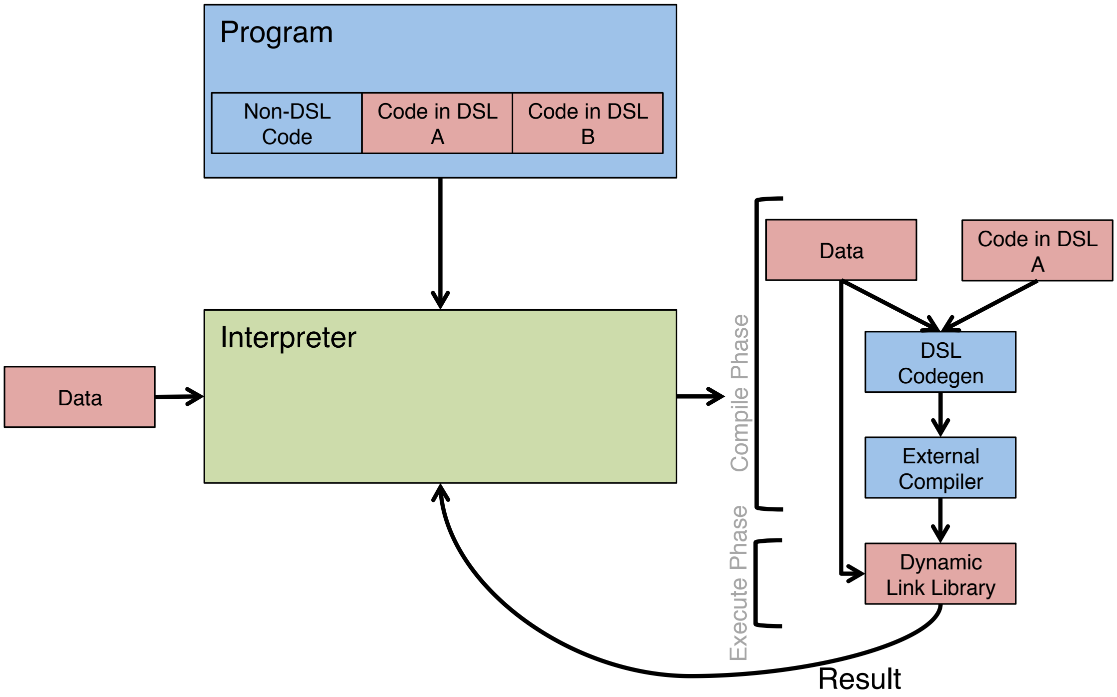


Figure 4.1: An overview of the SEJITS methodology. When a program containing DSEL code is run, the code triggers the interpreter to call a DSEL compiler (itself a set of libraries in the interpreted language) which dynamically generate external low-level code, which is compiled into a dynamic link library. This library is then called via the interpreter's foreign function interface with the input data, and results are returned to the interpreter. The compile phase only occurs if necessary; dynamically-created libraries are cached so that only the execute phase occurs after first run. Note that auto-tuning is elided for simplicity.

variants, until the best code variant is found, which is then always used.

Thus, in our running example, the portion of the code that defines computation on the graph is selectively and automatically transformed into a number of parallel variants in C++ and compiled; one of these variants is run in place of the high-level code defining the computation. To the programmer, however, it appears that the entire application ran in the high-level language interpreter.

SEJITS combines two separate features that exploit abilities of modern high-level "scripting" languages: embedded domain-specific languages, and runtime code generation with auto-tuning. In particular, SEJITS exploits the ability of modern scripting languages to introspect themselves, call external programs, and interface with external libraries using a highly-capable Foreign Function Interface (FFI). Along with these mechanisms, the methodology suggests a set of best practices that make the DSELs more useful for productivity programmers and easier to write for efficiency programmers. Together, these mechanisms and best practices form a new point in the design space of DSELs, enabling highly productive programming for non-expert programmers using languages they are already familiar with, as well as enabling performance programmers to productively build reusable DSEL compilers.

By separating out the roles of domain scientists and efficiency programmers, SEJITS enables a separation of concerns that allows each programmer to write code suited for the level they are most familiar with, as shown in Figure 4.2. In our running example, this means that the domain scientist only need worry about the correctness of their application; efficient execution of the graph algorithm is ensured by the performance expert who designed the graph DSEL. In addition, efficiency programmers can interface with third-party optimized libraries when necessary instead of needing to implement everything themselves. This is useful when existing libraries already do a good job of tuning the calculation, such as Intel's MKL [54] for dense matrix operations, FFTW [39] for Fast Fourier Transforms, or OSKI [115] for sparse matrices.

The target region for domain-specific embedded languages with the SEJITS methodology is shown in Figure 4.3. Although embedding the DSLs into a high-level language may incur overheads that could be avoided by writing in a hand-optimized low-level language, the resulting performance is far better than using the high-level language alone. Furthermore, the productivity benefits— in particular, the reduced lines of code due to separating optimization strategy from the implemented computation— may make the source slightly longer than a pure high-level language. However, the benefits of embedding plus code generation outweigh these possible downsides.

4.2 DSELs and APIs in Productivity Languages

Much of the previous work in domain-specific embedded languages concentrated on languages designed to be extensible using metaprogramming, including Haskell and variants of Lisp. These DSELs generally transformed host language code into other host language code, all running within the language interpreter or compiler [71].

Widely-used productivity languages today have good metaprogramming capabilities, but may not be well-suited for using these facilities for creating DSELs. Languages such as Ruby and Smalltalk have been extensively used to create DSELs [48, 49] while others such as Python are rarely used for DSEL creation. None of these languages have first-class macro systems. This lack of macro programming makes it more difficult to create non-trivial DSELs.

The SEJITS methodology adds better DSEL support to such languages by utilizing the class

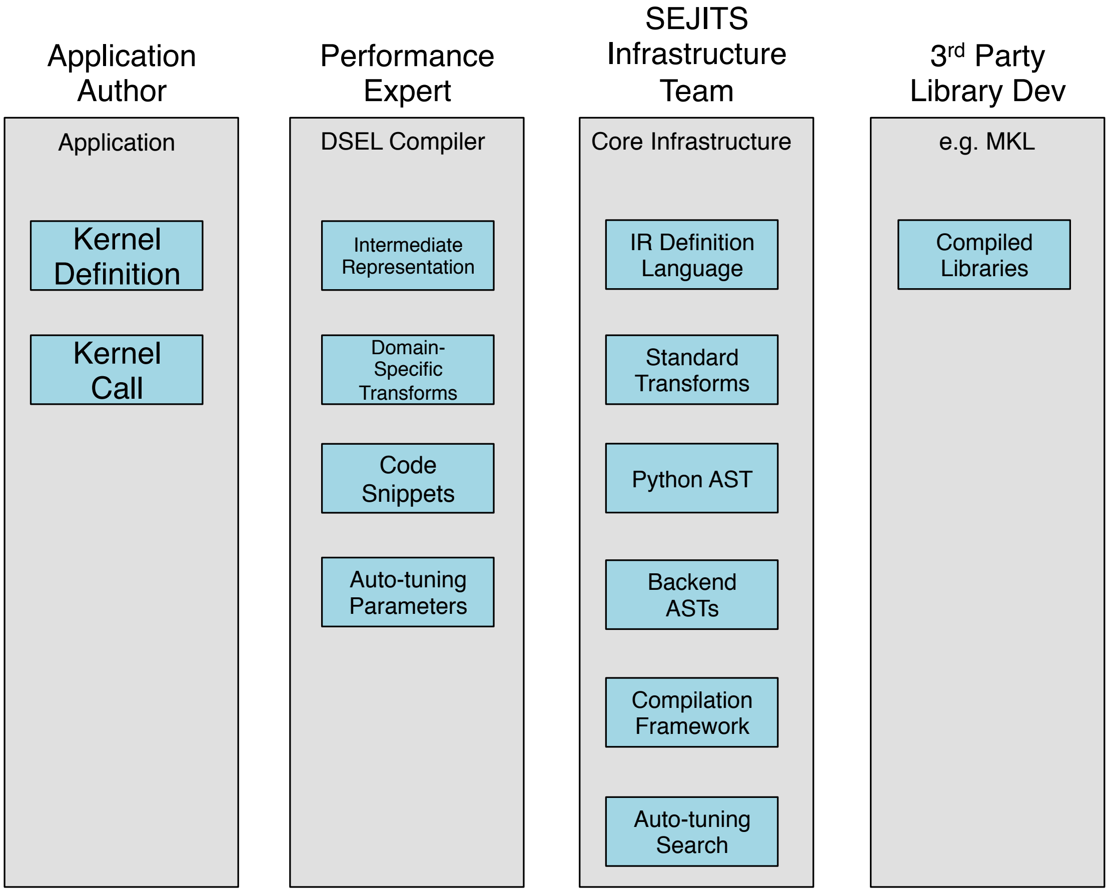


Figure 4.2: Separation of concerns enabled by SEJITS. Application writers need only write their applications to use the DSEL compilers (specializers), while performance experts restrict their concern to domain-specific aspects of code generation. SEJITS infrastructure can be leveraged to implement some general transforms and can abstract away compilation and caching of compiled code. Efficiency programmers can also interface with existing low-level libraries which do not have to be changed for use with SEJITS.

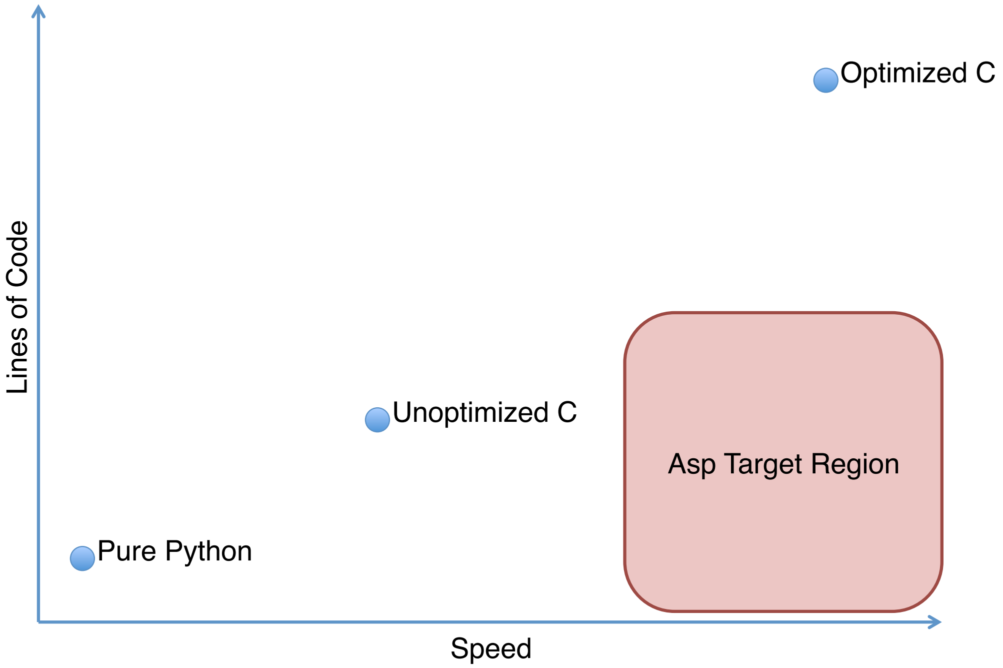


Figure 4.3: Target region for SEJITS. The methodology trades some potential performance for the ease-of-use of using embedded DSLs, but in return promises high performance at relatively lower lines of user code.

(object-oriented) systems in these languages to encapsulate the metaprogramming transformations required for each DSEL. Classes can be modified to parse selected functions using capabilities already existing in such languages, and these parse trees can be manipulated by a SEJITS system.

Furthermore, encapsulation via class points to a simple way to ensure domains are small— a class only needs to implement a domain-specific compiler for languages in its domain. Thus, the implementation and interfaces for users appear to be extensions to the class hierarchy, similar to libraries in modern productivity languages.

In our running example, the domain scientist designs their application to utilize a specific class in order to execute the graph algorithm portion of their code; this class inherits from the correct encapsulating class provided by the performance expert. In addition, the domain scientist properly defines any computations in this class that are necessary for correct code generation. Such restrictions are presented to the domain scientist in the DSEL documentation.

In contrast to Application Programming Interfaces (APIs), DSEL compilers do not necessarily follow the execution rules of the host language. Indeed, DSELs we implement are internally represented as declarative constructs, even though the programmer uses imperative syntax in the DSEL. For example, DSELs such as ActiveRecord [48] that implement Object-Relational Mapping (ORM) allow users to write code in the host language to query a database for objects, but the semantics of the code follow SQL's semantics. Thus, DSELs where the body of a function defines a series of operations in the embedded language are more appropriately thought of as declarative embedded languages, since they do not necessarily execute with the same execution model as the host language interpreter but instead use the computation as written as simply a specification.

4.3 Code Generation

Modern productivity languages are well-suited for manipulating text, and include support for regular expressions and other textual manipulations. Combined with the ability to write files and call external programs, these capabilities allow non-trivial code generation and compilation using external compilers. In the SEJITS approach, DSELs generally leverage external optimizing compilers, since this enables reusing both the expertise of existing efficiency programmers (who are used to writing code in these low-level languages such as C++ and CUDA) and reusing the expertise embedded in the compiler (which has been tuned to highly optimize code).

Code generation in itself would be useless without the Foreign Function Interfaces in modern productivity languages. These FFIs allow programmers in the productivity language to call libraries written in C or C++ , as long as those libraries implement the correct interface and link against the correct libraries. Importantly, most productivity languages allow loading arbitrary FFI-enabled libraries during program execution. SEJITS leverages these FFIs to load freshly-compiled (or cached) libraries into the running interpreter and execute them; this hides the compilation and load steps from domain programmers.

A major reason for using the FFI and low-level languages is to support parallelism, which is often unsupported by these productivity languages due to thread-unsafe interpreter structures. For example, Python uses a Global Interpreter Lock (GIL) to ensure only a single thread is executing, partially in order to ensure interpreter data structures do not become corrupted. External FFI-loaded libraries do not have such limitations, as most interpreters disable garbage collection while external code is running. Thus, for many productivity languages, the only path to reasonable parallelism is to encapsulate parallel execution in FFI-loaded libraries.

Just-in-time code generation has some advantages as well. Because at call-time the information about the data in the call is complete, dynamically generating the code allows specializing the code for the particular data. This can be as simple as inlining loop bounds or as complicated as introspecting matrix structures to see if calling symmetric-specific routines is appropriate. Although runtime code generation does incur the overhead of producing the code, the code it produces can be highly optimized with information specific to that execution.

Most importantly, code generation at runtime provides information that can limit the search space for auto-tuning over the many possible ways to implement a calculation. For example, in an auto-tuned SEJITS compiler for matrix operations, characteristics of the matrix could guide what kinds of optimizations are possibly beneficial.

In our running example, the code generated for the graph algorithm is executed via the FFI, and the results returned back to the overall program that is running in the interpreter. The particular code generated for the algorithm may be specialized for the class of graphs the domain scientist is using, thanks to runtime code generation.

The combination of DSELs and code generation is the most powerful use of SEJITS, but the two capabilities can be used separately as well. In fact, some DSELs may not generate external code but can instead generate Python code. Similarly, another use of SEJITS is for “trivial” DSELs that contain only a single operation, such as the Gaussian mixture modeling library (see Section 13.2). These trivial DSELs use the code generation and runtime auto-tuning capabilities of the methodology to obtain high performance while looking, to the productivity programmer, like normal libraries. In other words, the SEJITS approach is also a suitable mechanism for packaging auto-tuned libraries.

4.4 Auto-tuning

Compilers generally use complicated machine models and heuristics to attempt to generate optimized compiled programs, but previous work (see Section 3.1) has shown that they do not produce the best possible executables in many circumstances unless the user explicitly optimizes their code. In fact, different code "appearances" that implement the same algorithm result in different performance. Due to the difficulty of determining the right combination of explicit optimizations that results in highest performance, low-level programmers use *auto-tuning* to generate a large number of parameterized variants and then run them on representative inputs to find which variant gives the best performance. Subsequently, this best-performing variant is used.

Because these large-scale compilers with many man-years of work cannot optimally generate code, we do not expect that our small DSEL compilers will always be able to do so. Although the small DSEL compilers have domain knowledge, enabling them to perform optimizations general compilers cannot safely do, the output of our DSELs (i.e. low-level code) is dependent on the external compiler for much of its performance. We therefore treat auto-tuning as a first-class capability in SEJITS; many DSELs using the SEJITS methodology use auto-tuning and SEJITS systems try to make auto-tuning as simple as possible.

Because in SEJITS compilation occurs at runtime, much of the auto-tuning is runtime auto-tuning. In this approach, a DSEL compiler produces a number of variants for the calculation, and each time the calculation is to be performed, a different variant is run, with historical performance data preserved. Deciding in which order to explore variants can be done using a variety of methods (such as hill-climbing, gradient ascent, or genetic algorithms). After all variants have been exhausted or the auto-tuning system determines no more should be tried, subsequent executions use the fastest variant.

Some variants for our running example may perform the graph algorithm slower than others, but after running the program numerous times, whether for validation or through the course of development, the variants have been pruned such that the fastest ones have been found. Subsequent runs of the prototype application will always use this optimal variant.

The combination of code generation and auto-tuning is powerful and allows DSEL compilers in the SEJITS approach to be less dependent on complicated machine models or heuristics. Runtime auto-tuning provides empirical data for obtaining high performance.

4.5 Best Practices for DSELs in SEJITS

In addition to the mechanisms described previously, the SEJITS approach includes best practices for DSELs. These conventions help make the user experience of using DSELs better for the productivity programmer and help efficiency programmers more easily create embedded languages.

One major convention is that any valid host language program should run, even if the compiler infrastructure does not yet support something the productivity programmer is attempting to write. For example, if a programmer uses a valid Python construct, but the compiler requires it to be expressed differently, the experience of having the program unexpectedly not run is frustrating. Instead, the convention is that the DSEL itself should be valid Python, and if the compiler cannot produce compiled code from the user's source, the source itself runs directly in the interpreter. The user is issued a warning (ideally showing how the code should be modified) and the execution still occurs, albeit slowly.

To ease development of DSEL compilers, we believe they should be *analysis-avoiding* as much as possible. That is, instead of relying on analysis to ensure user-supplied code is correct or to determine how to translate to the lower-level language, the embedded language should force the user to specify the necessary information, within reason. In other cases, additional information available at runtime can be used in lieu of static analysis; since there is little distinction between compile-time and runtime in a dynamic language, this information is available when code generation occurs. By eliminating the need for most analysis, SEJITS should enable faster development of DSELs.

4.6 Language Requirements to Enable SEJITS

In order to support this combination of DSELs and runtime code-generation/auto-tuning, candidate hosting languages for SEJITS must support a number of mechanisms in the language interpreter or runtime.

Specifically, languages need to support the following capabilities:

- *Writing files*. Because SEJITS relies on external optimizing compilers, host languages must be able to produce files to pass to the compiler.

- *Calling external programs*. DSEL compilers invoke external compilers, so the host language needs to support this invocation, inheriting any necessary environment variables or search paths used by the compiler.

- *Parsing functions*. Ideally, a host language should be able to parse (or return a parse tree for) a function by name. When this is not possible, an equivalent capability would be to detect where the currently-running file is located and run a parser (external or internal to the runtime) that returns the necessary parse tree.

- *Foreign Function Interface*. Host languages must support the ability to interface with functions written in the lower-level language, including support for manipulating data. Without this capability, data must be serialized and passed between them using other mechanisms, resulting in poor performance.

- *Loading arbitrary shared libraries from user-writable paths*. If the FFI does not support runtime loading of arbitrary libraries, it is difficult or impossible to support seamless execution of the newly-compiled code. Some environments, such as the Java Virtual Machine, do not allow loading from arbitrary paths, and require users to set environment variables before starting the JVM that dictate from where libraries can be loaded. Although this is less than ideal, it can be worked around in SEJITS implementations.

The SEJITS methodology can still be implemented in the absence of some of these capabilities, although it may require extensive workarounds. However, many modern productivity languages support these features, including Ruby [102], Racket [93], Python [100], and Lua [109]. Other languages, such as Scala [110] and Groovy [46], which run on the JVM, require some minimal workarounds (for example, restricting code generation paths to known directories).

4.7 Summary

This chapter described the SEJITS methodology and outlined some of its salient features. In addition, we see that a small set of best practices can ensure the best environment for non-expert programmers without a huge amount of investment from developers. Many modern scripting languages can use the SEJITS approach, which combines code generation/auto-tuning with domain-specific embedded languages, bringing advantages from both.

Chapter 5

Asp is SEJITS for Python

In this chapter, we describe the infrastructure we have built in Python for implementing SEJITS domain-specific embedded auto-tuning compilers. We begin in Section 5.1 with an overview of Asp (Asp is SEJITS for Python). We illustrate Asp functionality using a trivial example DSEL in Section 5.2. We then describe in more detail the facilities provided by Asp. In Section 5.3 we describe the intermediate representation used in Asp, and Section 5.4 describes code generation capabilities. Debugging support for developers of DSELs is outlined in Section 5.6 and auto-tuning support is in Section 5.7. Section 5.8 summarizes.

5.1 Overview of Asp

We have built an infrastructure library called Asp for developing SEJITS auto-tuned DSL compilers embedded in Python, using our strategy of generating code for an external compiler that is then run using the foreign function interface. We choose Python as the target language for pragmatic reasons: there is a large amount of momentum for the language, with many scientific programmers using it as a replacement for Matlab by utilizing the SciPy and NumPy libraries [57, 88], which together provide similar functionality as Matlab. Furthermore, the language supports the mechanisms required for our approach, as outlined in Section 4.6.

A DSEL compiler in our approach generally consists of phased transformations that implement the following steps at runtime (leaving aside caching for the moment):

- parsing program text into a Python Abstract Syntax Tree (AST),
- transforming the Python AST into a *Semantic Model*, an intermediate form expressing the semantics of the domain-specific computation in terms of supported language constructs,
- optionally applying architecture-independent optimizations to the semantic model,
- then transforming the Semantic Model into an output AST for the target language,
- optionally optimizing the computation expressed in the output AST,
- generating program text from the output AST, including many possible versions if using auto-tuning,

- and finally, compiling the generated code into a dynamic link library that is loaded into the host language using its Foreign Function Interface (FFI) and run, returning the result to the program while recording performance.

The overall flow is shown in Figure 5.1, with auto-tuning elided for simplicity. Note that many of these steps can be cached for use in subsequent invocations, and such a cache is almost necessary to amortize the overhead of compilation.

Asp provides functionality for obtaining Python ASTs from the interpreter, transforming ASTs and Semantic Models, defining Semantic Models, generating code in supported backend languages, compilation for a number of backend toolchains, as well as caching and calling previously-compiled DSEL code, and auto-tuning. It builds on a number of existing Python libraries, and treats commonly-used numeric collection types from the NumPy and SciPy libraries as built-in data structures, making it easy to use them in DSELs.

5.2 Walkthrough: Building a DSEL Compiler Using Asp

In this section, we build a trivial DSEL compiler that performs very limited kinds of map operations over lists of integers. A map operation applies a user-supplied unary function to each item of a list or array. The DSEL we implement will allow expressing arithmetic map computations such as those shown in Figure 5.2. Although the DSEL itself performs a trivial function, it demonstrates some of the features of our approach as well as the features Asp provides for DSEL writers.

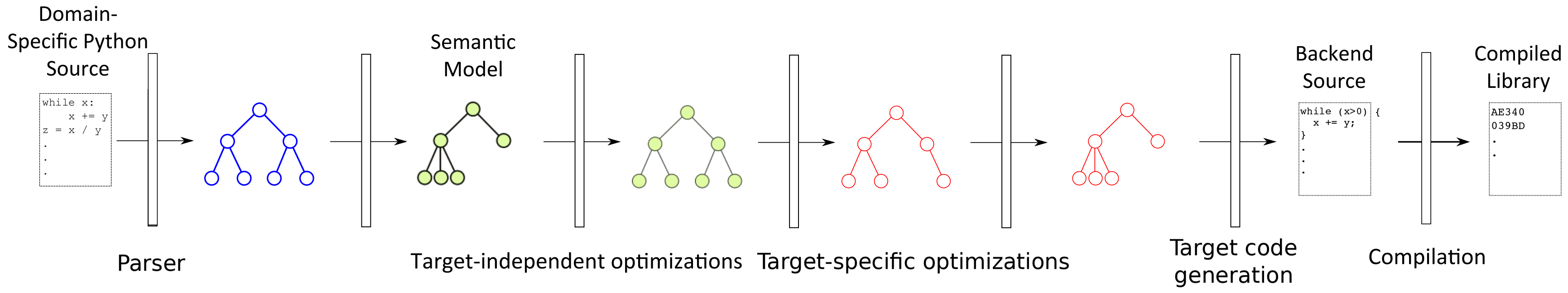
The figure illustrates the interface for DSEL end-users: the user creates a new class, subclassing from the DSEL's `ArithmeticMap` class, and writes the computation on each element as an instance method `f(x)`. The user then applies the custom function to each element of a list of integers using the instance method `map()`. This use of standard Object-Oriented Programming (OOP) to express computations is one of the characteristics of Asp DSELs.

Note that the code in Figure 5.2 is valid Python; that is, in the absence of any DSEL compilation, a suitably-defined `ArithmeticMap` class would ensure that the Python code executes normally. In this case, all the pure-Python `ArithmeticMap` class would have to do is call Python's built-in `map()` using the user-defined `f()` function.

We will now more formally specify the kinds of computations the DSEL will be able to express; this is done by defining the intermediate representation, which we call the Semantic Model after Hudak [52]. The goal of a Semantic Model is to express the set of computations the DSEL implements. The user's Python code is transformed into instances of the Semantic Model, and in doing so, is turned into a structure that the later steps in the compilation flow can generate backend code for.

5.2.1 Defining the Semantic Model

In Asp, the Semantic Model intermediate representation is a tree structure that defines the operations allowable in a DSEL. Though the declaration of the Semantic Model looks similar to defining syntax, it is used to concretely describe a particular instance of DSEL use, not to define the syntax of that use. Put another way, the Semantic Model is a way of describing a declarative intermediate representation that, once defined, restricts what computations can be generated by the DSEL; an


Domain-Specific Python Source
Parser
Semantic Model
Target-independent optimizations
Target-specific optimizations
Target code generation
Backend Source
Compilation
Compiled Library

```
while x:
    x += y
z = x / y
.
.
.
```

```
while (x>0) {
  x += y;
}
.
.
.
```

```
AE340
039BD
.
.
```

Figure 5.1: Stages in Asp transformation from user-supplied program text into obtaining the result of the computation.

```
from arithmetic_map import *

class Squarer(ArithmeticMap):
  def f(self, x):
    return x * x

class DoublePlusOne(ArithmeticMap):
  def f(self, y):
    return (y * 2) + 1

Squarer().map([1,2,3,4])
# result: [1,4,9,16]

DoublePlusOne().map([1,2,3,4])
# result: [3,5,7,9]
```

Figure 5.2: Examples of end-user code that utilizes the DSEL we build in this section.

instance of the Semantic Model is similar to a general compiler's intermediate representation of an instance of computation.

Our DSEL expresses basic arithmetic expressions by allowing the user to write a pure unary function (one with a single input, since each item of the input list gets its own function invocation, and no side effects). The function applied at each item is restricted to using integers and basic arithmetic operations, to limit the implementation difficulty for this example. The Semantic Model for ArithmeticMap is shown in Figure 5.3, written in Asp's integrated DSEL for defining Semantic Models.

```
# Top-level node.  We keep track of the input identifier to the unary function.
MapFunction(ReturnExp, input_id=String)

# The computation must return something
ReturnExp(Expr)

Expr =  Integer
      | InputIdentifier
      | BinaryOp

# A binary operation here is one of +,-,*,/ and can use either the input
# or an immediate integer
BinaryOp(left=Expr, op={ast.Add, ast.Sub, ast.Mul, ast.Div}, right=Expr)
```

Figure 5.3: Semantic Model for our simple ArithmeticMap DSEL, defined using Asp's integrated embedded DSL for the intermediate representation.

Although the example Semantic Model is quite simple, it illustrates several principles of designing SMs. First, the goal of the definition is to, as much as possible, make instances of the Semantic Model "correct-by-construction;" that is, an instance that conforms to the SM definition will always be a correct instance in the sense that the DSEL compiler will generate correct code for it. Secondly, the definition itself is written in our embedded DSL for Semantic Models, which

```
class ArithmeticMapSMBuilder(NodeTransformer):
  # turn the function definition into a MapFunction SM
  def visit_FunctionDef(self, node):
    assert len(node.args.args) == 2, "The number of arguments to f must be 2"
    assert type(node.args.args[1]) == ast.Name
    self.input_id = node.args.args[1].id
    return MapFunction(self.visit(node), input_id=self.input_id)

  def visit_Return(self, node):
    return ReturnExp(self.visit(node.value))

  def visit_Name(self, node):
    assert node.id == self.input_id, "Unknown identifier %s" % (node.id)
    return InputIdentifer(node.id)

  def visit_Num(self, node):
    return Integer(node.n)

  def visit_BinOp(self, node):
    return BinaryOp(self.visit(node.left), op, self.visit(node.right))
```

Figure 5.4: Code to convert a Python AST expressing a computation in our DSEL to a Semantic Model instance.

makes it simple to define Semantic Models and to check their correctness— after each instance of an SM is constructed during the initial phases of runtime translation, it is type-checked to make sure the nodes match restrictions expressed in the definition. More details about the embedded DSL for expressing Semantic Models are in Section 5.3.

After defining the Semantic Model, we must express how computations written using Python syntax will be transformed into instances of the model.

5.2.2 Transforming Python to Semantic Model Instances

The top-level control for our DSEL compiler passes the parsed syntax tree for the function `f()` to a tree transformer that performs node-wise transformation of the AST into an instance of the Semantic Model. This is done using Asp's `NodeTransformer`, which extends built-in Python functionality to create a unified tree transformation framework for Python ASTs, Semantic Models, and backend ASTs.

The transformations use the visitor pattern [92]. In this design pattern, DSEL compilers define transformations by defining *visitor functions* for each node type; the infrastructure then calls the appropriate node visitor function when visiting a particular node. Our code for transforming a Python AST for `f()` to a Semantic Model instance is shown in Figure 5.4.

When converting the Python AST to a Semantic Model instance, we check any assumptions the Semantic Model structure makes before converting the appropriate node. For example, the number of arguments to the DSEL function `f()` must be two (following the Python convention that instance methods have `self` as their first input). Similarly, since the only identifier allowed in the computation is the input to `f()`, the transformer keeps track of this identifier and ensures that no

```
class ArithmeticMapCppTransformer(NodeTransformer):
  def visit_MapFunction(self, node):
    return cpp_ast.FunctionBody(FunctionDeclaration(cpp_ast.Value("int", "f"),
                                                    cpp_ast.Value("int",
                                                                  node.input_var)),
                                [self.visit(node.body)])

  def visit_ReturnExp(self, node):
    return cpp_ast.ReturnStatement(self.visit(node.value))

  def visit_InputIdentifier(self, node):
    return cpp_ast.CName(node.id)

  def visit_Num(self, node):
    return cpp_ast.CNumber(node.value)

  def visit_BinaryOp(self, node):
    op_map = {ast.Add: "+", ast.Sub: "-", ast.Mul: "*", ast.Div: "/"}
    return cpp_ast.BinOp(self.visit(node.left),
                         op_map[type(node.op)],
                         self.visit(node.right))
```

Figure 5.5: Backend code generator for the user-expressed `f()` function in our ArithmeticMap DSEL.

other ones are used. If any assumptions are violated, DSEL compilers should give concrete, useful feedback to the user as to why the computation cannot be compiled; such feedback is elided from the example here for brevity.

For our simple DSEL, the transformer in Figure 5.4 is sufficient, but for larger, more expressive DSELs, much of the DSEL implementer's effort goes into deciding the interface for users (with emphasis on making sure the interface is expressive, intuitive, and respects Python semantics) as well as ensuring users receive useful feedback when their computations do not conform to DSEL restrictions.

5.2.3 Generating Backend Code

The next phase of DSEL compilation takes a Semantic Model instance and converts it into backend code, using a combination of templates (code in the output language with "holes" that are filled in) and tree transformations. The transformer that turns our ArithmeticMap `f()` function into a C++ function, again using the visitor pattern, is shown in Figure 5.5. Because we have already ensured the constructed Semantic Model is restricted to what we can correctly compile, this code is relatively straightforward.

For each node in the Semantic Model, equivalent nodes for a C++ AST are generated by our transformer. Some of the assumptions about the transformed function are encoded in this class; in particular, the C++ function always has the signature `int f(int <input_var>)`. This allows us to include templated "glue" code (not shown) that handles unpacking Python arrays, applying the function `f()` to each element, and returning the result.

An example of the output from our DSEL compiler is shown in Figure 5.6. In addition to the

```
#include "Python.h"

// generated from our user-supplied function
int f(int x) {
  return (x * 2) + 1;
}

// generated from an Asp template
PyObject* map(PyObject* in) {
  // code that unpacks the python array and
  // applies f() to each element, returning a new array
}

// generated boost::python code for exposing the functions
BOOST_PYTHON_MODULE(module) {
  boost::python::def("f", &f);
  boost::python::def("map", &map);
}
```

Figure 5.6: Generated C++ code from DoublePlusOne using the ArithmeticMap DSEL.

generated function and the templated glue code, there is also support code that uses the Boost Python library [1] to expose C++ functions to Python. This support code is automatically generated by the Asp infrastructure.

Finally, all that is required is some simple top-level code that orchestrates translation and compilation of the user-specified code, shown in Figure 5.7. The class first checks to make sure the user defined the required `f()` method, then attempts to compile the specified function. If it succeeds, the Python method is replaced by the method in the Asp module, which will run the C++ version.

Although this DSEL is relatively simple, it follows the general flow of a number of DSELs we have implemented using the SEJITS approach and demonstrates the different pieces of functionality Asp provides for DSEL writers. One major aspect of our DSELs that is not shown in this example is the use of auto-tuning; in that case, instead of adding a single generated function in the top-level code, the DSEL compiler writer adds a number of variants, each with its own program text, optionally with functions that determine which of the variants are valid for which inputs. The Asp infrastructure then will handle auto-tuning over the variants.

In the rest of this chapter, the components of the Asp infrastructure are specified in more detail.

5.3 Expressing Semantic Models

Asp provides a built-in DSEL for defining Semantic Models, which we used in Figure 5.3. Based on the DSEL creator's definition of the Semantic Model, our DSEL creates and runs Python code that defines Python classes for each node in the Semantic Model. Within each class, automatic type-checking code ensures that only classes matching the user's specification can be used.

Users can also specify additional semantic checks to be inserted into the generated classes. For example, the specification:

```
MyNode(args=Identifier*)
```

```
class ArithmeticMap(object):
  def __init__(self):
  # make sure the method f() is defined
  if not self.f:
    raise Exception, "Method f() must be defined."

  # obtain the Python AST and try translating
  try:
    ast = asp.parse(self.f)
    sm = ArithmeticMapSMBuilder().visit(ast)
    cpp_f = ArithmeticMapCppTransformer().visit(sm)

    # create a new Asp module to hold the compiled function
    self.module = asp.jit.AspModule()
    self.module.add_function("f", cpp_f)
    self.f = self.module.f

 except:
    # if we can't compile, we just use the interpreter as usual
    print "Warning: Unable to compile instance.  Using interpreted version."
    pass
```

Figure 5.7: Top-level code for the ArithmeticMap DSEL.

```
  assert (len(args) == 2 or len(args) == 0)
```

would insert the specified assert statements into the initialization code for the `MyNode` class. This allows DSEL creators to easily encode checks while still benefiting from our infrastructure.

The output Semantic Model classes are subclasses of an Asp-provided node class, which ensures they can be transformed or visited using a unified visitor class. These visitor classes mimic Python's built-in `ast.NodeVisitor` and `ast.NodeTransformer` classes, but extend the ability to write visitors and transformers to Semantic Model classes as well as output ASTs. The interface for writing such visitors is straightforward: DSEL writers create classes that subclass the correct visitor and specify methods of the form `visit_<classname>(self, node)`. Our infrastructure then automatically dispatches the correct method when encountering a node of a particular type. For transformers, the method must return the replacement node; for visitors that do not transform the Semantic Model or AST, the return type is ignored.

5.4 Code Generation

Asp provides two paths for generating code: templates (backend code with "holes" and Python control sequences) or using Abstract Syntax Tree nodes. Currently, the two paths are not unified in the sense that Asp does not translate from templates (or arbitrary source code) into AST nodes. However, templates can include AST-generated code snippets. Most DSELs use combination of the two techniques. Templates are ideal for code that remains relatively static in different computations, such as the "outer-loops" of iterative calculations, as well as glue and utility code. The Asp template language uses the Mako [8] syntax, which consists of backend output code as well as Python control structures. This combination is actually quite powerful, and can be used for non-trivial templates. For example, the following template code generates a fully-unrolled dot product function over two vectors:

```
void dot_product(float* vec1, float* vec2, float* output) {
  *output = 0.0;
% for x in xrange(vector_len):
  *output += vec1[${x}] * vec2[${x}]
% endfor
}
```

Note that the vector length needs to be passed in at the time the template is rendered into C++ code.

Generation using the output language ASTs is most useful when the user defines the computation; this will be true for most DSELs of interest. Using C++ ASTs in Asp also brings some advantages: Asp includes some built-in optimization routines (such as applying loop unrolling or blocking to an AST) that can be used by DSEL implementers, and Asp's unified tree transformation framework can be used to write additional, domain-specific optimization routines. The C++ AST is based on CodePy's `cgen` library, but contains many more nodes and implements the proper protocol to allow our tree transformation machinery to work.

A number of alternative code generation approaches were explored during the development of Asp. We considered using the Weave framework (part of SciPy [57]), which attempts to allow interspersing of C++ and Python code and automatically compiles the native code, while interfacing with Python. However, Weave did not support a big enough subset of the language. Similarly, projects such as Cython [11] enable writing C (and recently, C++) library wrappers in a Python DSEL, but code generation is limited to translating some Python expressions into C. We also explored using the AST implementation from `libclang`, the library that implements LLVM's Clang [69] compiler for C/C++ /Objective C. At the time we were developing Asp, the Python bindings for the AST were very limited. Some progress has been made, but the AST implementation is still relatively low-level and supports only C-like languages. Nevertheless, Asp could be targeted to use `libclang` for C and C++ code generation and compilation; this would allow Asp to benefit from optimizations in the LLVM toolchain, but may make it more difficult to interface with other compilers.

In addition to ISO C++ , Asp also includes limited support for generating CUDA and OpenCL ASTs, and Scala support is in progress as well. Table 5.1 shows the various languages and level of support in Asp.

5.4.1 Dealing with Types

Because Python is dynamically typed while variables in our backend languages are statically typed, during code generation it is necessary to determine the types of inputs to program text that is being generated. Often, determining types in unnecessary, such as when the DSEL compiler only operates on data of a particular type (for example, a matrix computation DSEL may only operate on double-precision floating point matrices).

In cases where types must be determined, DSEL compiler writers can use run-time introspection for finding the concrete type of data. In addition, Asp provides a trace-based type analysis that runs the code to be transformed in the interpreter, while tracking concrete types of all left-hand

Language	AST Support	Template Support	Compilers
C++	yes	yes	GCC, LLVM-GCC, Intel CC, clang, MSVC
Cilk Plus [55]	yes	yes	Intel CC
CUDA [87]	partial	yes	NVCC
OpenCL [63]	partial	yes	Intel CC
Scala	partial	yes	Scala 2.9

Table 5.1: Asp language support and supported compilers.

sides of assignment statements. Although this analysis only provides concrete types for a single execution, this can be sufficiently useful for the DSEL compiler to help determine backend types for code generation, especially when the same objects are repeatedly operated on (such as in iterative methods).

5.5 Just-In-Time Compilation of Asp Modules

Asp provides functionality for creating functions in many languages, using many backends. Asp's interface for adding functions to be compiled is that the functions are contained, logically, in an Asp module (like a Python module). Different functions in an Asp module may belong to different backends, and the infrastructure attempts to abstract away the backends when functions are called.

Adding a function to Asp involves specifying the function text (as a rendered Asp template, or as a backend language AST) and the appropriate backend to use for compilation. Additionally, Asp allows specifying *callback policies* which control how objects and return values are returned to Python from the compiled function. By default, Asp attempts to use Boost translators for C++ elemental types, which automatically translate between Python types and C++ types when passing data between the two. If C++ structures or objects are being returned, users can specify which parts of the C++ class should be exposed, and whether Python's garbage collection or backend language memory management should be used.

To limit the time calling compilers (which are usually not optimized for speed), previously-compiled code is cached. In addition, Asp compiles functions lazily, triggering backend compilation only when a function in the backend is called, and only if function text has changed since the cached version was created. Caching and compilation functionality is mostly provided by CodePy [65], the support library used to build Asp's JIT support, but we have added lazy compilation (waiting until call-time to compile) for efficiency reasons.

One limitation of Python's foreign function interface (FFI) is that the compiler used on the native code accessed through the FFI must be compatible with the compiler originally used to compile Python. On some platforms, this limits the compiler toolchains Asp can support. Table 5.1 shows the compilers and languages supported by Asp's JIT compilation machinery.

5.6 Debugging Support

To aid DSEL compiler writers, Asp includes some basic debugging capabilities. Two kinds of debugging are currently implemented [123]: an implementation of parallel race detection and a two-level stepwise tracing, to ensure the equivalence of interpreted and generated code.

The parallel race detection is a variant of NDSeq [21], and is used to detect parallelism bugs introduced in the optimization stage of a DSEL compiler. Our backend code generator for C++ can be configured to automatically instrument parallel execution constructs (such as OpenMP loops) to check for these kinds of bugs. The instrumentation checks for potential races and then records execution, which is then fed into the NDSeq serializability checker.

The stepwise trace algorithm allows DSEL implementers to check correspondence between the generated and pure Python code, for finding bugs in code generation. To use this, DSEL compiler writers implement a Python interpreter for their Semantic Model, and use the debugging infrastructure to instrument corresponding execution points between the Semantic Model and the output code. For example, loop iterations that must be executed in some order can be instrumented in both the Semantic Model and the output code. Then, the tool checks and reports any violations where values differ between the two.

5.7 Auto-tuning Support

Asp provides support for auto-tuning by allowing multiple variants of a function to be created when passing a function to Asp for compilation. In addition to each variant, DSEL implementers can pass in functions that determine, based on properties of the inputs, whether a particular variant can run. At call-time, a function with a number of variants is first checked to see which variants have not yet run, then that list is filtered by which variants can run given the input, and finally, a variant is picked. However, if all variants have been run, the fastest runnable one is always picked.

Enabling this functionality is a persistent database that records timing information for variants run on input data; in addition to the variant, properties of the data specified by the DSEL writer are recorded as well. Asp implements both a local database as well as support for a global database implemented as a software service queried via RESTful HTTP calls [37]. However, this latter functionality is not yet enabled.

There is currently no explicit support for install-time tuning, as used in libraries such as pOSKI [56]. Install-time tuning runs many variants at install-time to determine choices at run-time; for example, pOSKI runs many implementations of sparse matrix vector multiply at install-time, then uses the performance numbers to choose, at run-time, based on a performance model, the right implementation to use. Although Asp does not have explicit support for this, install-time tuning can use the same database infrastructure and populate the performance database with relevant data at install-time; the database interface is self-contained and simple, and as such can be easily integrated with other packages.

Currently, the default auto-tuning search strategy uses exhaustive search without ordering. DSEL compiler writers can override this to provide their own search strategies. Future improvements to the search will allow DSELs to use more intelligent search algorithms, such as hill climbing, gradient ascent, or genetic search algorithms [5].

5.8 Summary

Asp, our infrastructure for implementing SEJITS domain-specific compilers in Python, aims to ease the difficulty of writing DSEL compilers that use external compilers and the language's foreign function interface to interact with the interpreter. This chapter demonstrates this functionality using a simple DSEL, and outlines some of the available infrastructure for DSEL writers.

Chapter 6

Experimental Setup

Throughout the case studies that follow, we use three machines of varying structure to demonstrate the portability and high performance of each of our DSELs. In this chapter, we outline the experimental platforms and methodology for the three major case studies in the thesis. Section 6.1 describes the hardware platforms for our experiments. In Section 6.2 we describe the programming environments on our machines. Section 6.3 outlines the performance measurement methodology of the thesis. Section 6.4 summarizes.

6.1 Hardware Platforms

For the three major DSELs in our study, we use three different hardware platforms with quite different characteristics. One of the machines represents workstation-class architectures while the other two are server-class machines from Intel and AMD. Details of the three machines are shown in Table 6.1 and discussed in the rest of this section.

Postbop is a workstation-class machine with an Intel Core i7 870 processor with four cores in a single socket. This processor is from the Lynnfield generation of processors and is built on a 45 nm process, with a TDP of 95 Watts. Although it is capable of using Intel's Turbo Boost technology to temporarily scale up processing speeds when not all cores are utilized, our test machine has this feature disabled in order to maintain consistent benchmarking performance. Postbop has 8 GB of memory and can achieve about 17 GB/s sustained memory bandwidth and 23.4 GFlop/s double precision compute rate. Note, however, that the memory bandwidth figure requires using more than one core; tests indicate a maximum single-core memory bandwidth of about 14.6 GB/s.

Boxboro represents an Intel server-class machine and contains four Intel Xeon E7 4860 processors, codenamed Westmere-EX. Each processor contains 10 cores, for a total of 40 cores in this machine. Built on a 32 nm process, these processors have a massive 24 MB shared L3 cache and can sustain a measured memory bandwidth of up to 66.5 GB/s, thanks to their quad-channel QuickPath interconnect. Peak computing rate is 181.6 GFlop/s in double precision. The TDP of each processor is 130 Watts. For our test machine, we disable Turbo Boost in order to maintain consistent timing measurements. This large scale machine allows us to test our codes on a large single-node shared memory architecture. Note that the presence of four sockets, each with their own integrated memory controllers, means that some applications will experience Non-Uniform Memory Access (NUMA) effects, due to accessing memory that is allocated on a different socket.

Machine Name	Processor	Clock (GHz)	Socket	Cores (w/HT)	L2 (KB)	Shared L3 (MB)	Memory (GB)	Peak Mem BW (GB/s)	Peak DP Compute (GFlop/s)
Postbop	Intel Core i7 870	2.93	1	4 (8)	256	8	8	17.0	23.4
Boxboro	Intel Xeon E7-4860	2.27	4	40 (80)	256	24	128	66.5	181.6
Hopper	AMD MagnyCours	2.1	2	24	512	6	32	52.7	201.6

Table 6.1: Machines used for experiments. Peak memory bandwidth and compute rates measured empirically using microbenchmarks. Note that Hopper's MagnyCours architecture consists of two six-core processors per die, each with 6 MB of shared L3 cache each.

Machine	Kernel	Python	Compilers	MPI Runtime
Postbop	Ubuntu Linux 3.0.0	Ubuntu 2.7.2+	Intel CC 12.0.2 GCC 4.6.1	MPICH2 1.4
Boxboro	Ubuntu Linux 3.0.0	Ubuntu 2.7.2+	Intel CC 12.1.0 GCC 4.6.1	MPICH2 1.4
Hopper	Cray Linux Env 4.0	2.7.2	GCC 4.6.1	Cray

Table 6.2: Software versions used in this study.

Hopper is a large-scale Cray XE6 supercomputer at the National Energy Research Scientific Computing Center (NERSC), run by Lawrence Berkeley National Laboratory for the Department of Energy. Each of the 6,384 nodes contains two 12-core AMD MagnyCours processors running at 2.1 GHz, but each of these processors is made up of two separate on-die 6-core processors. Each 6-core processor shares a 6 MB L3 cache. Each node has a measured peak memory bandwidth of 52.7 GB/s, and a compute rate of 201.6 GFlop/s. The supercomputer uses a custom Cray interconnect and has a collective compute capability of 1.28 PFlop/s. Because of the unique node architecture, there are four different Uniform Memory Access (UMA) domains of 6 cores each; like Boxboro, application performance may suffer from NUMA effects.

6.2 Software Environment

This section describes the software environment on our test machines, including compilers and parallel programming models used. Table 6.2 summarizes the software environments.

6.2.1 Compilers & Runtimes

On all the Intel machines, we use Ubuntu Linux 11.10 with the latest kernel patches; this means the kernel is Ubuntu's modified/patched version of the Linux 3.0.0 kernel. On all machines, we use 64-bit Operating System installs and utilize compilers that conform to the x86-64 Application Binary Interface (ABI). On Hopper, we use Cray's custom version of Linux, Cray Linux Environment 4.0. This uses a heavily modified version of Linux kernel 2.6.32 with custom Cray patches to minimize variability and reduce the amount of services running on the compute nodes.

On the Intel machines, we use Intel's C++ compiler version 12 due to its excellent ability to optimize code for the platform. For codes using the Message Passing Interface, however, we use the default MPI compiler on the platform, which is built on Gnu Compiler Collection (GCC) version 4.6.1. For Hopper, we use Intel's C++ compiler for single-node tests, since it gives the best performance when we compare installed compilers. For multi-node tests, we use Cray's MPI implementation built on top of GCC.

6.2.2 Parallel Programming Models

In our DSELs, we primarily use two different programming models for backend code: OpenMP and MPI. OpenMP [89] is a set of C++ pragmas and a library for multicore parallelism on shared memory machines. Users of OpenMP annotate their programs with pragmas that define how loops or parallel sections should be executed. It is the primary mechanism we use for our loop parallelism constructs. OpenMP is supported by all the compilers we use, and we use features from OpenMP 2.0.

MPI is the Message Passing Interface and is the standard programming model for high performance computing applications running on multi-node machines. Although the programming model can be used for non-SPMD (Single Program Multiple Data) parallelism, we use it strictly in an SPMD manner; in other words, all of our MPI programs run the same program on all nodes. Individual cores run the program with memory in their own address spaces; any communication between cores occurs explicitly through two-sided messaging. In addition, various runtimes can build implementation-specific optimizations for shared memory. We use MPICH2 [47] on Intel machines and Cray's highly optimized MPI implementation on Hopper.

6.3 Performance Measurement Methodology

6.3.1 Timing Methodology

Many of our experiments use auto-tuning, which involves running a number of variants of the same code. In most cases, we will report the best-timed variant. All timing is done multiple times (at least 5) and we report the mean performance, after eliminating the first run since it often incurs first-run performance differences. For measurements where cache effects could make subsequent runs faster than expected (due to some data still being present within the cache) we take care to clear the cache between each timed run by running some untimed kernel that uses different data.

6.3.2 Roofline Model

When evaluating the performance results, we will employ the Roofline [119] model as an empirical method to determine how far we are from the best possible performance. The Roofline model is a two parameter visual model that can be used for prediction, modeling, and analysis in a kernel-specific and machine-specific way.

The Roofline model defines *operational intensity* as the ratio of compute operations (often, this is floating-point arithmetic operations) to total DRAM memory traffic. Operational intensity is a measure of the characteristics of the execution of a kernel on a specific machine. In contrast, *compulsory operational intensity*, defined as the ratio between computation operations and compulsory memory traffic, is a measure of the characteristics of the kernel itself; that is, it uses the minimum possible memory traffic, which is usually measured in terms of the memory size of the data structure.

In the model, the attainable performance is defined as

$$\text{performance} = \min\{ \begin{array}{l} \text{Peak Operational Performance} \\ \text{Peak Bandwidth} \times \text{Operational Intensity} \end{array}$$

In the first case, the kernel is said to be *computation-bound*, since the limit of performance is how fast the processing units can compute results. In the second case, the kernel is *memory bandwidth bound*. We can plot these two limits on the same axes and determine, based on the operational intensity, where the expected limit of a given kernel is on a given machine. The point where the operation computation limit and memory bandwidth lines meet is called the *ridge point*, and denotes the minimum operation intensity required to attain peak computational performance.

The full Roofline model adds ceilings based on characteristics of the memory and computation of a kernel on a machine. For example, if the floating point instruction mix is such that a given kernel can only obtain 80% of peak floating point performance on a machine due to instruction stalls or other issues, then that represents a lower ceiling.

Figure 6.1 show the machine rooflines for our test machines. Note that the ridge points are quite different on the three architectures: on Postbop, an operational intensity of only 1.5 is required for peak performance, while on Hopper the required operational intensity is nearly 4. This gives some insight as to what kinds of algorithm implementations will attain peak computational performance, and which will be bound by memory bandwidth. By plotting a vertical line at the operational intensity point as well as a point representing the actual performance attained, we can gain insight as to which type of optimizations are necessary: memory bandwidth or computational optimizations.

6.4 Summary

In this chapter, we outlined the machine and software characteristics of the three test machines we use in our major case studies. We described the Roofline model, used to determine what fraction of peak performance we obtain and what kinds of optimizations will bring us closer to this peak. Using the model, we can see that the three test machines have different balance between computational and memory traffic capabilities, and thus, we expect them to obtain different performance.

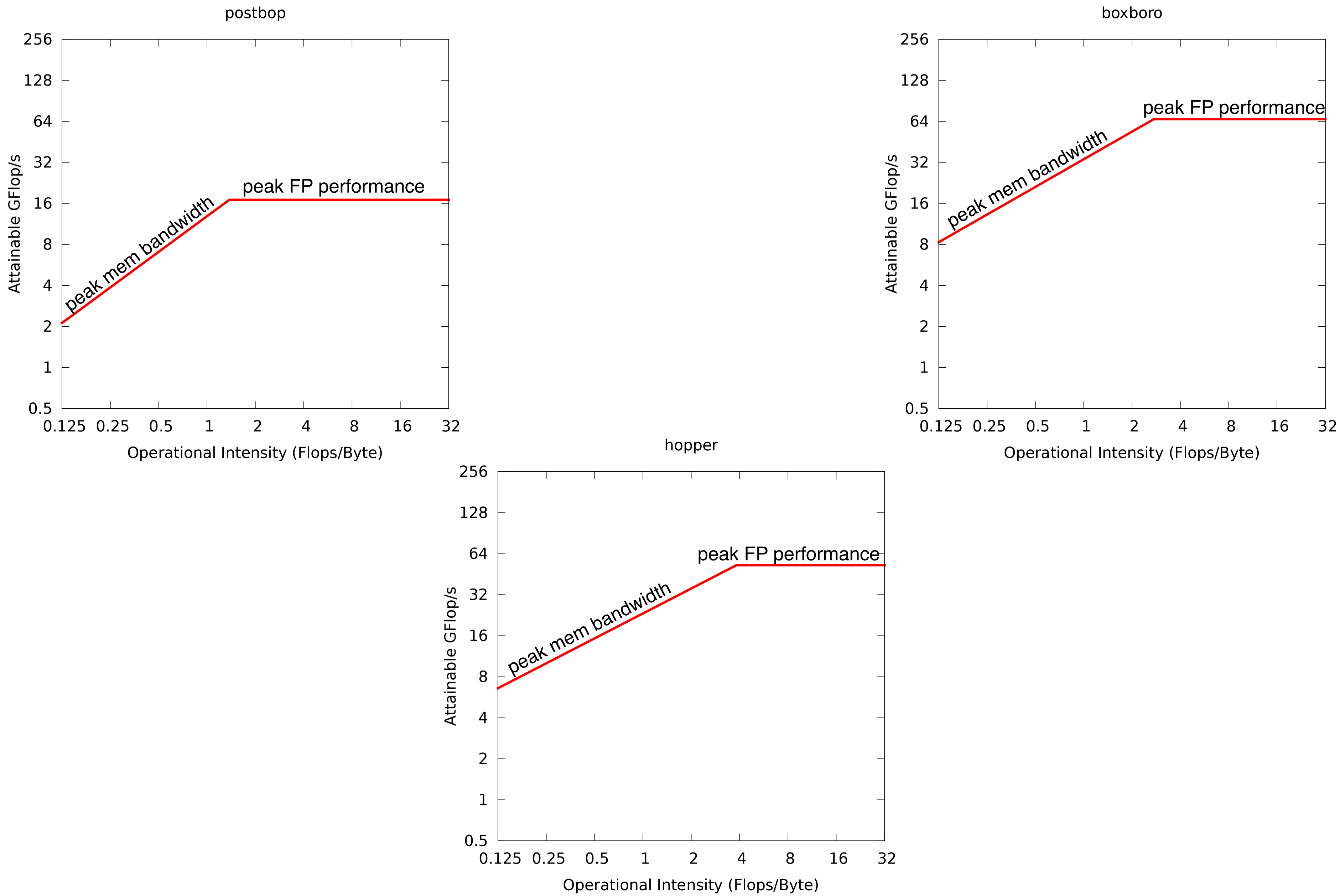


Figure 6.1: Machine-specific rooflines for the three test machines in this thesis, showing the relationship between peak attainable memory bandwidth and peak double-precision floating-point operation rates.

Chapter 7

Overview of Case Studies

In the following chapters, we outline case studies that implement domain-specific embedded languages using the Selective Embedded Just-In-Time Specialization (SEJITS) approach. These DSELs are embedded in Python and utilize the Asp framework described in Chapter 5. The different DSELs span a number of domains, and demonstrate different aspects of the approach. Table 7.1 summarizes the case studies and the important aspects of the SEJITS approach and the Asp framework highlighted by each. As we will see, the DSEL implementations are relatively small in terms of lines of code, pointing to the productivity of using Asp as a base infrastructure across the variety of domains.

In this work, we examine three major case studies of our own in two domains: a structured grid DSEL and two DSELs for the Knowledge Discovery Toolbox (KDT) [74] graph algorithms package. The first case, for structured grids (Chapters 8-10), demonstrates the advantages that DSELs bring to domains where traditional high performance libraries cannot effectively optimize computation due to the use of higher-order functions. Using the SEJITS approach, we build a DSEL that allows users to express their computations in a high-level way yet still obtain performance that is as good as or better than the state of the art. To do this, we use domain-specific knowledge coupled with code generation, transformation, and auto-tuning.

The goal of the graph algorithm DSELs (Chapters 11-12) is to mitigate the performance overheads of using a high level within KDT to allow users to productively analyze large, distributed graphs. Without the DSELs, large overheads are incurred due to the need to cross the boundary between native code (where the compute engine executes) and interpreted code (where user-defined code executes). With the filter DSEL, users can perform algorithms on subsets of semantic graphs without incurring huge overheads, and the DSEL for semiring operations allows advanced users to write new graph algorithms in the framework without resorting to C++ . These two DSELs demonstrate how the SEJITS approach and Asp can be used to mitigate the productivity-performance gap for existing applications that use high level languages, in addition to showing that the SEJITS approach can integrate well with distributed computation.

Besides these three major case studies, we summarize one other DSEL and two auto-tuned libraries developed by others using the Asp infrastructure and the SEJITS approach (recall from Section 4.3 that auto-tuned libraries can be seen as “trivial” DSELs from the perspective of SEJITS). First, we examine the matrix powers implementation (Section 13.1), which applies many optimizations using templated code to this sparse matrix kernel that is the essential building block of communication-avoiding Krylov subspace solvers [81]. We also explore the Gaussian Mixture

DSEL	Ch	Domain	Description	Features Highlighted
Stencil	8-10	Structured Grid	Auto-tuned DSEL for Structured Grid calculations	Code transformation, auto-tuning
KDT Filters	11-12	Graph	DSEL for on-the-fly filters for semantic graphs in the Knowledge Discovery Toolbox	Code transformation, distributed computation, integration with existing packages
KDT Semiring Operations	11-12	Graph	DSEL for enabling new algorithms by defining KDT semiring operations in Python	Code transformation, distributed computation, integration with existing packages
Matrix Powers ($A^k x$)	13.1	Sparse Matrix	Auto-tuned implementation of the $A^k x$ kernel for use in communication-avoiding solvers	Auto-tuning, template-based code generation
Gaussian Mixture Modeling	13.2	Machine Learning	Auto-tuned GMM implementation for GPUs and CPUs	Auto-tuning, template-based code generation, input-based CPU/GPU selection
BLB	13.3	Machine Learning	DSEL for using the Bag of Little Bootstraps (BLB) algorithm	Mixing template-based code generation and transformation, Scala support, cloud computing support

Table 7.1: Overview of the SEJITS case studies in this work.

Modeling (GMM) [82] implementation (Section 13.2), which also uses advanced code templates to generate data-dependent implementations of the algorithm on multicore CPUs and GPUs, while choosing on which platform to execute based on availability and problem parameters. Finally, we summarize an in-progress DSEL for the Bag of Little Bootstraps [64] machine learning algorithm (Section 13.3), which combines code templates and code transformation to execute user-provided code in parallel. Depending on problem size and availability, a Scala-backed version using the Spark [125] framework executes on the cloud, demonstrating the applicability of our infrastructure to cloud computing as well as local parallelism.

With these six examples, we demonstrate that Asp and the SEJITS approach enable delivering high performance while still programming in a high-level language. The variety of domains and backends demonstrate the broad applicability of our approach.

Chapter 8

Structured Grid Computations

Structured grid computations refer to a computational pattern in which each point in a multidimensional grid is updated with a function of some subset of surrounding data points. Such computations are used in a large number of applications, across many application domains, including linear algebra, imaging, and direct physical simulations. In this chapter, we discuss the structured grid motif, including properties of computations in this domain, as well as optimization strategies and performance models.

A structured grid computation is usually performed on a multidimensional grid, with a specific function applied at each point. The surrounding points involved in the function are called *neighbors*, and collectively they define the *shape* of the computation. In many cases, the computation is different for the *interior* points than it is for *boundary* points, which are points that are located on the grid's logical boundaries and may have different computations applied depending on the specific application.

The motif of structured grid computations is interesting in that though the applicable optimizations are well-known, how and when to apply these optimizations is not established. Structured grid computations are challenging for auto-tuning due to the fact that the function applied at each point, properties of the grid it is applied on, and the information at each point are all application-specific. There has been much work in optimizing such computations, either with compiler optimizations or with auto-tuning. Compiler optimizations suffer from needing to use heuristics and models combined with static analysis to determine what optimizations are applicable, correct, and yield highest performance.

Auto-tuning optimizations have generally been applied only to a single kernel instance at a time by using text manipulation scripts to generate many variants of a kernel. This is due to the fact that the computations are application-specific. Moreover, the optimizations themselves are dependent on properties of the structured grid computation. This makes it impossible to package the result of auto-tuners as a universal library.

This chapter proceeds as follows: we outline some characteristics of structured grid computations in Section 8.1 and how they map to constructs and operations on computers, as well as data structures in Section 8.2. Section 8.3 describes optimizations for these kinds of computations and Section 8.4 describes some performance models. Finally, Section 8.5 summarizes.

8.1 Characteristics of Structured Grid Computations

The structured grid motif is prevalent across many domains in computer science, and such computations differ from one another in a number of ways. In some cases, the computations represent simulations of physical phenomena, but in other cases, the computations do not have a physical representation. This section outlines some canonical applications of structured grid kernels, and explores some of the characteristics of these kernels.

8.1.1 Applications

The canonical example of a mathematical structured grid operation is finding a direct solution to a system of partial differential equations (PDEs) representing Poisson's equation ($f = \bigtriangledown x$) using finite differencing. In this method, called Jacobi's method, each point on the grid represents a quantity and is updated using a weighted average of its neighbors until convergence. For a 2D rectangular grid, the complexity of the algorithm is $O(N^2)$ and for 3D it is $O(N^{5/3})$, where N is the total number of points in the grid. In Jacobi's method, the structured grid computation is the same as multiplying the vector that represents the state of the grid by the Laplacian matrix, which is encoded by the shape and weights in the stencil.

Convergence properties of such computations can be improved in a number of ways. In particular, two improvements to the Jacobi algorithm, Gauss-Seidel iteration and Successive Over Relaxation (SOR), can improve the solve to $O(N^{3/2})$ and $O(N^{5/3})$ for 2D and 3D, respectively, and are discussed in Section 8.3.

Mathematical solves are not the only application of structured grid algorithms. Canonical image processing algorithms used to analyze, blur, or change the image to make it more amenable to feature extraction all use structured grid kernels.

In the next sections, we define and characterize different parameters that specify an instance of a structured grid calculation: dimensionality (Section 8.1.2), connectivity (Section 8.1.3), and topology (Section 8.1.4).

8.1.2 Dimensionality

One major difference between structured grid kernels is the *dimensionality* of the grid, which often corresponds to a physical dimensionality, though in some cases the physical geometry is mapped to a higher-dimension grid when the connectivity between points is complex. Dimensionalities of 1D to 4D are common, but higher-dimensional grids are sometimes used. Dimensionality also influences what kinds of connectivity and topology can occur.

Figure 8.1 shows the structure of one type of grid as the dimensionality is increased, demonstrating 1D, 2D, and 3D rectahedral grids. As is clear in the figure, dimensionality influences the structured grid computation because it helps determine how many neighbors are present for each point in the grid.

8.1.3 Connectivity

The *connectivity* of a grid describes how individual nodes in the grid are connected to other nodes. For example, in a rectangular 2D grid, each node is connected to its four or 8 neighbors, depending

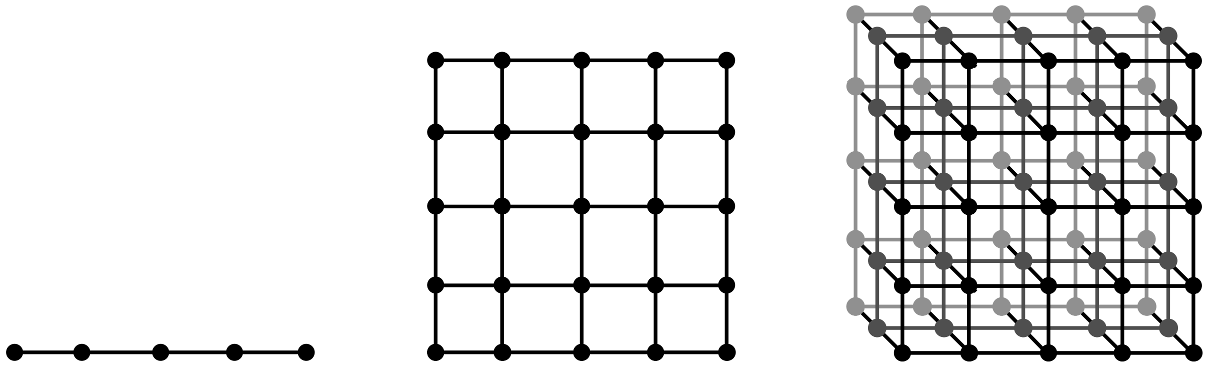


Figure 8.1: Left to right: 1D, 2D, and 3D rectahedral grid structures. Different dimensionalities also influence possible connectivity. Dimensionality is one of the major ways different structured grid kernels differ.

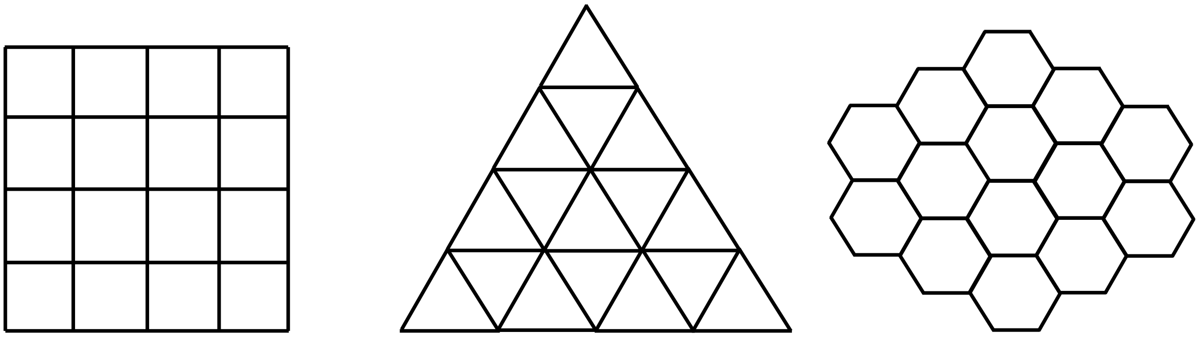


Figure 8.2: Left to right: rectangular, triangular, and hexagonal connectivity for 2D grids. Connectivity, along with dimensionality, dictates which points are considered to be neighbors of a central point.

on whether a neighbor is only along an axis or not. In contrast, a hexagonal grid in 2D has 6 or 12 neighbors. Clearly, in addition, the connectivity is not independent of the dimensionality, since the possible set of neighbors is influenced by dimension.

Examples of different connectivities for the same dimensionality are shown in Figure 8.2. Rectangular, triangular, and hexagonal grids in 2D are shown, each with a differing number of neighbors. In addition, another factor in determining connectivity is whether the values are taken to be *cell-centered*, *edge-centered*, or *node-centered*. Examples of the three for a hexagonal 2D grid are shown in Figure 8.3. Depending on the structured grid application, one or more of these may be used in different kernels, and the translation between the different types may require an application of a kernel that determines one kind of value from the other. For example, flow quantities may be most naturally expressed at edges, while the quantity under study exists at the center.

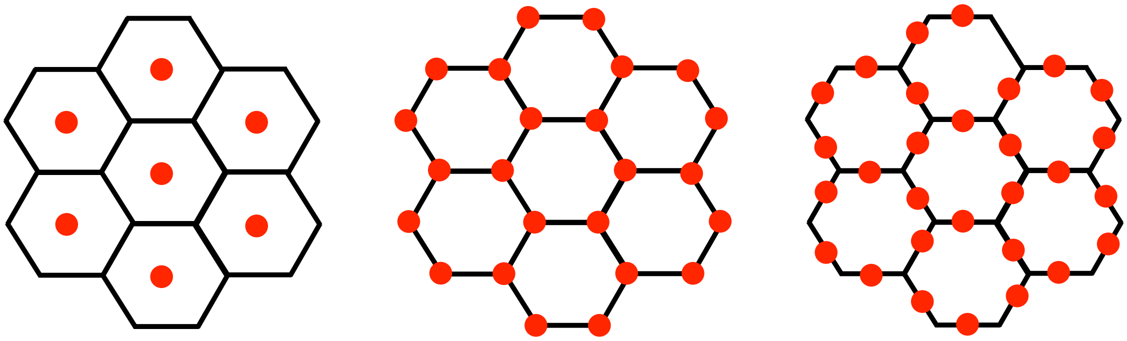


Figure 8.3: Left to right: cell-centered, node-centered, and edge-centered value locations for hexagonal 2D grids.

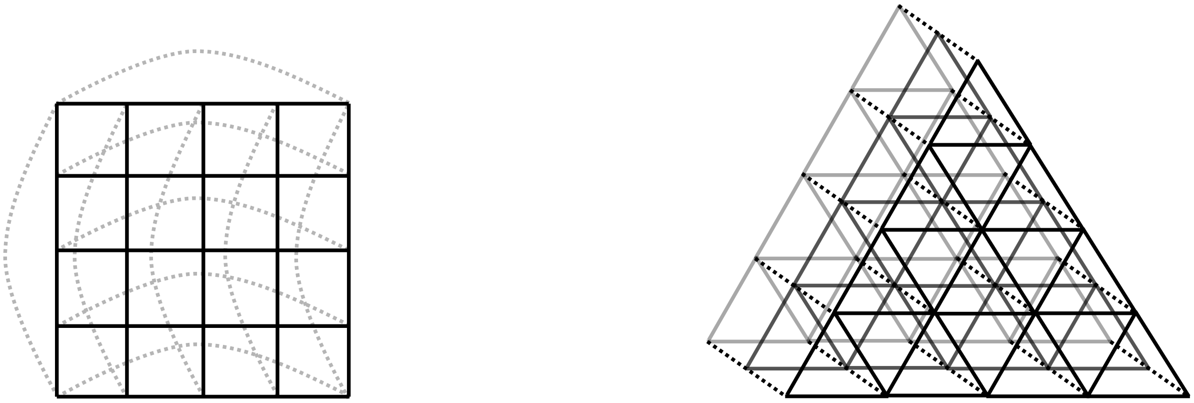


Figure 8.4: Example topologies for structured grids. Left: a 2D toroidal grid, with "end" connections shown with gray dotted lines. Right: a square pyramidal 3D grid, which consists of "layers" of triangular surfaces.

8.1.4 Topology

The *topology* of a grid refers to the combination of dimensionality, connectivity, and borders. For example, for a linear grid in one dimension, it is possible either have a 1D linear topology, or the ends can be connected together to form a torus. Similarly, a 2D torus and a 2D rectangle may both have rectangular connectivity, but differ in how the edges connect or do not connect to each other.

Figure 8.4 shows two example topologies. On the left is a 2D toroidal grid, with both edges connecting to the points on the other side, resulting in each point in the grid having exactly the same number of neighbors. The right part of the figure shows a 3D square pyramidal grid, which consists of triangular surfaces connected to the upper and lower surfaces; as a result, each cell has five axis-centered neighbors. Alternatively, a 3D tetrahedral grid would have all surfaces being triangles, resulting in fewer neighbors for each cell.

So far, we have only examined grids with a regular structure. Often, due to modeling physical phenomena, the grids will be a composition of two or more kinds of topologies, resulting in disparate regions where the connectivity and neighbors differ. For example, a rectangular and triangular grid may be composed by joining two grids. Such topologies require complicated mathematics to ensure that the boundaries correctly transfer and preserve physical quantities, and structured grid computations on these kinds of data structures may require different kernels for the two regions as well as an additional kernel for the boundary calculation.

8.2 Computational Characteristics

In this section, we examine how the characteristics of structured grid algorithms map onto data structures and computations on actual computer hardware. We will restrict most of the discussion to Jacobi-like structured grid kernels, where the output grid is distinct from input grid(s) to the kernel, and, for clarity, describe the straightforward computation only. A simple example of a structured grid calculation in C++ is shown in Figure 8.5 and demonstrates many computational aspects of such kernels. Optimizations are covered in Section 8.3.

```
void kernel(double* in_grid, double* out_grid, int nx, int ny) {
  // use a macro for index calculation
#define Index2D(_i,_j) (_i+nx*_j)

  // do the interior only
  for (int i=1; i<nx-1; i++)
    for (int j=1; j<ny-1; j++)
      out_grid[Index2D(i,j)] += 0.25 * (in_grid[Index2D(i+1,j)]
                                       + in_grid[Index2D(i-1,j)]
                                       + in_grid[Index2D(i,j+1)]
                                       + in_grid[Index2D(i,j-1)]);
}
```

Figure 8.5: Simple 2D structured grid kernel in C++ . Note that the data is stored as flat 1D arrays, even though the computation is on a 2D structured grid.

8.2.1 Data Structures

In general, many data items may be stored at each point in a grid, but for structured grid algorithms the number of data items (and hence the size of memory used) is constant per point. The data stored at each point may be a scalar value, a vector, or some larger data structure.

In the case of a simple scalar per point, a multidimensional grid is usually allocated as a single contiguous chunk of memory and indexed using integer offsets. In Fortran and languages that use Fortran-like storage, this is a simple multidimensional Fortran array, accessed using `grid(i,j)` for the i, j entry in a 2D grid, for example. In C, however, the lack of efficient multidimensional arrays results in the data structure usually being a single-dimensional C array that is then accessed using macro expansion or other indexing methods to calculate the offset from the base pointer. If the grid were instead implemented using C multidimensional arrays, multiple accesses to memory would be required, since a C multidimensional internally is a set of pointers to other pointers. Figure 8.6 shows the logical view, memory view, and indexing for C and Fortran for a simple 2D grid that stores scalar values.

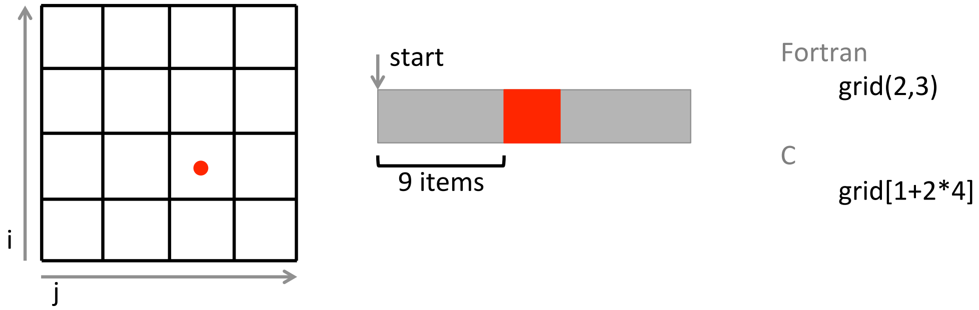


Figure 8.6: Left: Logical view of a 2D grid that holds scalar values, with a point highlighted in red. Center: View from the perspective of memory for the highlighted red location. Note that in memory, the grid is simply a flat 1D memory array. Right: Accessing the highlighted point in Fortran and C. Although C supports multidimensional arrays, the implementation is inefficient due to the large amount of "pointer chasing" incurred. Thus, structured grid computations usually mimic the Fortran model of multidimensional arrays by creating a flat 1D array and explicitly calculating offsets.

For grids that contain a vector or some larger set of values per point, two different implementations may be used. The most straightforward is to create an array similar to that in the scalar case, but where each item in the array is a C `struct` instead of a built-in datatype; this is called an *array of structures*. However, depending on the kernels performed on the grid, it is often more efficient to separate out the data items into separate arrays; for example, a grid of 3D vectors may be represented by three different arrays, one for each of the three components. Such a layout is called a *structure of arrays*. Both layouts for grids are common in structured grid computations, and each has possible tradeoffs in terms of ease-of-use and performance, depending on the kernels used.

8.2.2 Interior Computation & Boundary Conditions

In many structured grid algorithms, computation is divided into interior and boundary regions, because the operator applied to the interior can be done without dealing with special values or restrictions that occur at the boundary. Boundary values often require different computation structure, whether to match a physical restriction or to ensure mathematical correctness. Thus, the computation is often decomposed into two steps, so, on a grid G,

$$x = f_{interior}(x, neighbors(x)) \quad \forall x \in interior(G)$$
$$x = f_{boundary}(x, neighbors(x)) \quad \forall x \in boundary(G)$$

are done separately. Rarely, the boundary computations can be inserted into the same iterations as the interior computation, as in the case for periodic boundaries.

Common boundary conditions used in structured grid computations are fixed, periodic (corresponding to say, a torus or ring), and other types, which are generally implemented through the use of *ghost zones*. Fixed (also known as *constant*) boundaries are the simplest and most common. In this case, the boundary values are set before the structured grid computation and are never updated with new values.

Periodic boundaries occur when the logical connectivity of nodes at the boundaries are such that they connect to the other end. These can often be done along with the interior during a sweep of the grid, at the cost of having to use a integer modulus operator to ensure that values wrap around from the largest index back to the smallest in each dimension. The 2D toroidal grid in Figure 8.4 is an example where periodic boundaries occur.

Finally, some structured grid computations require more complex boundary calculations, including when the boundary must be recalculated before each sweep of the grid, or when the grid consists of two parts with different structures connected through a common boundary. Another common operation at the boundaries is to fill in data from remote computations when running a structured grid kernel in parallel. In many of these cases, the boundaries are implemented as *ghost zones* that are filled in via communication or computation that occurs outside the structured grid kernel. Ghost zones refer to cells that are not considered part of the interior of the grid and are not updated during a sweep, but are used as inputs to points on the edges of the grid.

8.2.3 Memory Traffic

Multidimensional structured grid calculations exercise the memory hierarchy in interesting ways, and many structured grid kernels are constrained in performance by the available memory bandwidth. In streaming through a grid, there is usually some spatial locality, but it may be difficult for the memory system to see it. One issue is that the spatial locality is difficult for the memory system to discern due to the way data is laid out in memory: accessing a 1D array using 3D addressing can prevent spatial locality from being obvious to the memory subsystem. As a result, unnecessary cache misses are often incurred in naïvely-structured implementations of structured grid kernels.

Prefetching engines are mechanisms in the memory hierarchy that track recently-loaded cache line locations and attempt to discern patterns from them and use these patterns to prefetch data from memory before it is requested by an explicit load. Prefetchers can eliminate the latency of waiting for a memory fetch to occur, but the typical memory layout of grids means the prefetchers must be intelligent enough to discern *strided* memory access patterns— consecutive points do not necessarily correspond to consecutive memory locations, for all but one of the dimensions in a multidimensional grid. On some architectures, these access patterns can confuse the prefetchers and prevent them from working effectively or even slow down the overall computation due to incurring unnecessary memory traffic if they fetch data that will not be used.

Some structured grid kernels require lookups into a table based on some function of values, or based on some function of the point locations to determine the coefficients to apply in the kernel. The first case is analogous to having different matrix values in a sparse matrix vector multiplication. The second one is sometimes used for non-rectangular connectivities, but can complicate the kernel in a way that is not amenable to high performance. Both uses of a lookup introduce memory latency delays into the function applied at every point; such latency can greatly increase processor stalls, as well as preventing vectorization (by hand or by the compiler), further impacting performance.

8.3 Optimizations

Depending on the particular kernel, structured grid computations are amenable to a wide variety of optimizations, some of which are suited for the case where the overall computation is bound by

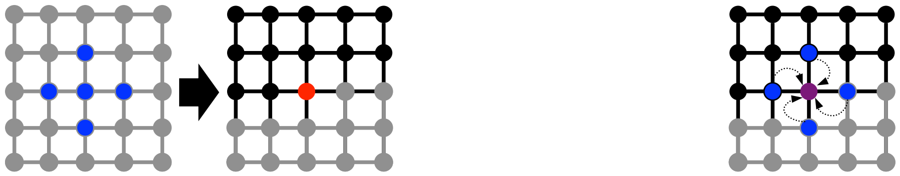


Figure 8.7: Left: Jacobi method on a 2D grid. The blue points of the input grid are inputs into the function that determines the output red point in the output grid. Note that all the blue points have "old" values (denoted by gray). Right: Gauss-Seidel method on a 2D grid. The same grid is used for input and output, so, as a result, some of the input blue points have old data and some have new data. The central point is used as both input and output.

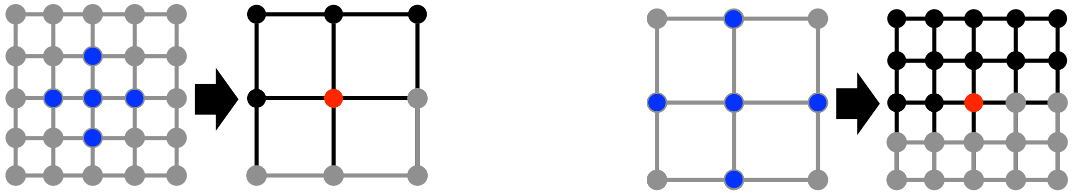


Figure 8.8: The restriction and prolongation operators used in multigrid, both of which are implemented as structured grid kernels. Left: Restriction operator on a 2D grid. Points in the coarser grid depend on fine grid points, with some kind of weighted averaging used in the translation. Right: Prolongation operator. Another set of weighted averages translates data from the coarse grid to the finer one.

memory bandwidth performance, and others for the case when it is bound by computation. This section outlines some of the optimization strategies prevalent for these kinds of computations.

8.3.1 Algorithmic Optimizations

Optimizations that change the algorithm (as opposed to just changing the implementation) can result in large performance improvements. The Gauss-Seidel method for structured grid algorithms, originally applied to solvers, uses a single grid for both input and output. Figure 8.7 compares the Jacobi and Gauss-Seidel methods. Since some of the values in the Gauss-Seidel method are new values, the propagation of information is faster, which results in faster convergence. For 2D, the convergence is improved to $O(N^{3/2})$ and similar speedups apply for other dimensionalities. However, the algorithm introduces dependencies between points, making parallelization difficult or impossible. A further improvement, called Successive Over Relaxation (SOR), weights the structured grid coefficients in a way that improves convergence without creating difficulties for parallelization.

Structured grid kernels are also the basis for another method of solving PDEs, called the multigrid method [18]. In a multigrid calculation, the values in the grid are *restricted* (i.e. represented by fewer points) and the computation is done on this coarser grid. Subsequent steps perform *prolongation* on the values to use more points to represent the grid, then do the computation on these finer grids. In this manner, convergence is improved to $O(N)$. The prolongation, restriction, and computation

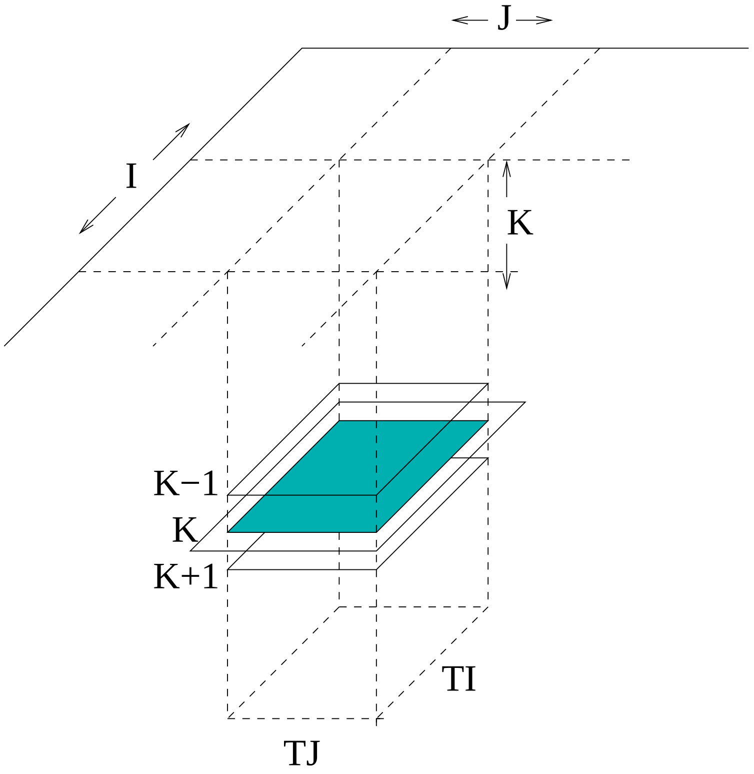


Figure 8.9: 2D blocking of a 3D structured grid problem. In this figure, K is the least-contiguous dimension in terms of memory layout. Rivera blocking only blocks in the other two dimensions (with blocking factors of TI and TJ), to prevent the tiny block sizes that result from a full 3D blocking.

steps are all implemented using structured grid kernels. Figure 8.8 shows a simplified example of restriction and prolongation for a 2D grid.

Adaptive Mesh Refinement [13] is a more advanced structure for such problems that uses insights from the multigrid method. It attempts to automatically decide on mesh fineness in regions that need more resolution, instead of increasing the fineness for the entire problem space, which can degrade performance substantially. The algorithm is too complicated to outline here, but can perform computations with accuracy that is not obtainable otherwise due to memory constraints. At its heart, the AMR method also uses structured grid kernels.

8.3.2 Cache and TLB Blocking

The cache blocking optimization attempts to eliminate cache capacity and conflict misses by operating on chunks of data at a time, and has been applied to a variety of motifs and codes [121]. Cache blocking can substantially improve performance for portions of code that are bound by memory bandwidth performance. In structured grid calculations, cache blocking is usually done not by changing the data structure, as is the case in other motifs, but by altering the traversal order to expose spatial and temporal locality.

Cache blocking has been applied to structured grid algorithms in much previous work [101, 122]. For 3D and higher dimensionalities, Rivera et al demonstrated that blocking in all the dimensions results in relatively small blocks (since the block must fit in a fixed-size cache) that do not effectively increase performance. As a result, they suggested 2D blocks for higher-dimensional grids, as shown in Figure 8.9. Such blocking schemes ignore the most-significant dimensions and perform blocking in 2D slices of the least-significant dimensions.

In addition, our previous work [60, 61] showed that it is important to think about unit-stride accesses when cache blocking because overall performance is dependent on prefetching engines in the memory hierarchy that are ineffective when long streams of unit-stride access do not occur. In addition, our empirical data showed that, for serial structured grid implementations, it is usually not advantageous to enable cache blocking in the least unit-stride dimension. The model used for this work is outlined in more detail in Section 8.4.

In parallel implementations on multicore machines, cache blocking occurs automatically if dividing up the grid in the correct manner. In other words, the standard parallelization schemes divide the grid into cache blocks and then distribute those cache blocks over the available processors. Additional levels of blocking for lower-level caches can be useful as well, in addition to ensuring the distribution of blocks respects the machine hierarchy.

Cache blocking can be done explicitly via loop transformations if given a blocking factor; the nested loops that implement a structured grid kernel are transformed using strip mining and loop interchange. Alternatively, cache blocking can be done using cache-oblivious algorithms [40], which use recursion to repeatedly subdivide the iteration space. However, cache-oblivious algorithms usually need to have some cache awareness [61] as to when to stop recursion; otherwise, the recursive function call overhead can incur more overhead than the time saved using the blocking. Both methods can minimize cache misses to near the theoretical minimum.

For compilers to optimize explicit loop-based traversals, the polyhedral model [122] is one approach. It uses the loop bounds and dependencies to define a polyhedron representing the iteration space along with restrictions on traversal order. Based on this representation, the approach uses heuristics to determine how to minimize memory traffic. In practice, it is difficult to automatically create the optimal version, but the polyhedral model represents a promising way to automatically optimize nested loops without relying on domain-specific knowledge.

8.3.3 Vectorization

For computation-bound kernels, vectorization is important to fully-utilize available computation capabilities and obtain maximum performance. Compilers must ensure the absence of pointer aliasing to determine vectorizability in structured grid kernels, but code is often expressed in ways that are not amenable to the needed compiler analyses (for example, the structured grid kernel function may use pointers as inputs, and determining that these pointers point to non-overlapping regions of memory requires full-program information). In other cases, the vectorizability is not apparent unless the iteration order is changed or common subexpression elimination is performed. Other kinds of kernels contain indirect accesses to grids or lookup tables, which correspond to scatter/gather patterns on vectors. Most SIMD instruction sets do not yet support this kind of vectorization, limiting the effectiveness of automatic vectorization for structured grid kernels.

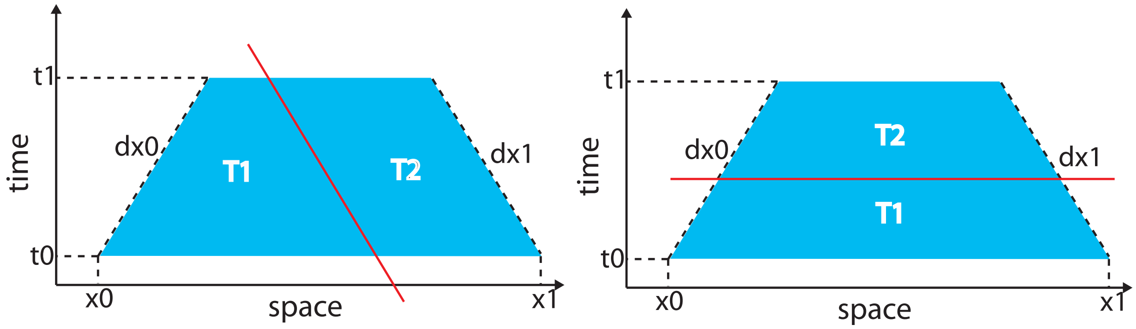


Figure 8.10: The serial cache-oblivious algorithm divides up space-time using recursive cuts. Left: a space cut in 1D space respects dependencies, so the space is cut at an angle relative to time. Right: a time cut can enable further recursive space cuts.

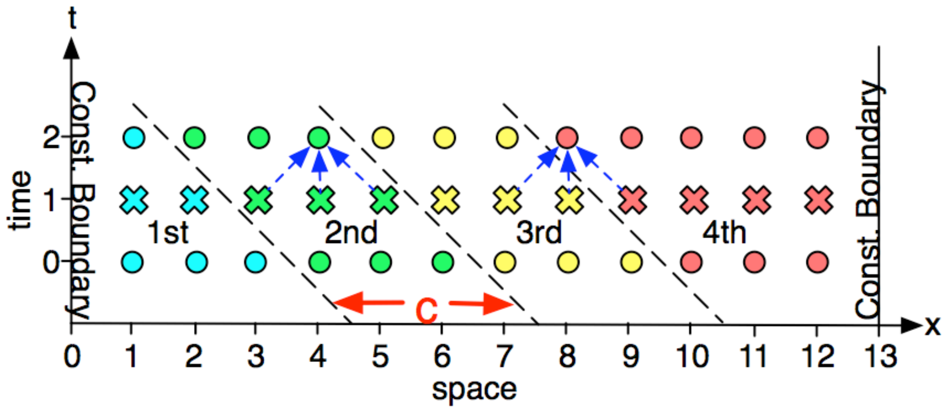


Figure 8.11: Time skewing for a 1D space stencil in 2D space-time. Same colored points represent blocks, where the blocksize is C. Blue arrows show, for two of the points, what points in the previous timestep they depend on. Note that some points depend on points in other blocks. Figure courtesy Kaushik Datta.

8.3.4 Locality Across Grid Sweeps

When the same kernel is applied repeatedly to a grid, which often occurs in iterative solver algorithms, it is sometimes possible to block in both space and time, that is, to operate on the same cache-sized block of data for several iterations at a time. This potentially eliminates capacity misses incurred by repeated sweeps, and can drastically reduce the required memory traffic for multi-step algorithms. However, it is sometimes non-trivial to change the computation order in this manner, due to dependencies between points both in space and time.

Blocking across multiple timesteps is the central insight that leads to time skewing [122] optimizations as well as the cache-oblivious algorithm [40], both of which perform blocking in the time and space dimensions. Figure 8.11 shows the time skewing optimization applied to a 1D kernel. Here, the blocks are in both space and time, with a space of C. Notice that some points depend on points in other blocks; however, if the computation proceeds serially, each point is updated for as many timesteps as the whole calculation requires before moving to the next block. Similarly, the cache-oblivious algorithm shown in Figure 8.10 uses blockings in space and time, but instead of explicit block sizes, it uses recursion to decrease blocks until they fit into cache. This recursion proceeds by trying to first cut in space (which reduces cache traffic) but, if the cut would yield a degenerate block, it proceeds to cut in time. Although the two algorithms are based on a similar principle, the performance characteristics can be quite different because time skewing is more amenable to compiler optimization and avoids overhead due to recursion [31].

8.3.5 Communication Avoiding Algorithms

Communication-avoiding algorithms use redundant work or other strategies to limit the amount of inter-unit communication in a parallel algorithm and communication between parts of the memory hierarchy in a serial computation.

The communication-avoiding matrix powers (or $A^k x$) algorithm [51] builds a basis $\{x, Ax, A^2x, ..., A^kx\}$ and is used in communication-avoiding Krylov Subspace Methods (KSMs) for solving systems of linear equations. The optimizations used in the algorithm are amenable to applying to multi-timestep structured grid algorithms. Conceptually, the communication avoiding optimizations in the structured grid context can be thought of as enhancements to space-time blocking. Instead of ensuring the blocked calculation proceeds in a way that respects point dependencies, the communication-avoiding strategy is to instead either perform redundant work (each block computes, from the first timestep, all points on which its values depend) or pre-compute and then communicate the pieces on which other blocks depend. In either case, the need for synchronization between the blocks is removed, allowing the calculations to proceed without repeated communications.

8.3.6 Parallelization

Structured grid calculations are amenable to both single-node (multicore) and multi-node (distributed) parallelism. In the single-node shared memory case, each hardware context is assigned a subset of the grid to operate on, possibly requiring (implicit) inter-thread communication for ghost zone data and for barriers to ensure correctness. In the distributed-memory case, explicit ghost zones that overlap between communicating nodes are used to ensure all the data required is present. Between iterations, explicit communication is required to send the values in ghost zones to appropriate processors. Parallelization can potentially speed up the computation by a factor of P if the kernel is compute-bound. For memory-bandwidth bound kernels, parallelization can help maximize obtained memory bandwidth.

Other optimizations may influence the parallelization strategy. For example, if cache blocking is employed, the parallelization must be aware of this blocking in order to obtain best performance. Similarly, the communication avoiding algorithms are different in serial and in parallel, with different goals. Thus, parallelization cannot necessarily be applied independently of other optimizations.

8.3.7 Summary of Optimizations

This section has described some optimization strategies and issues for structured grid computations. There is a long history of applying these optimizations to individual kernels and applications, with varying effectiveness. However, applying them generally is difficult due to their dependence on aspects of the individual computational kernel's properties, such as the footprint of the computation

Optimization	Depends on
Algorithmic Opts	Problem domain
Cache/TLB Blocking	Grid size, footprint, cache size, parallelization
Vectorization	Footprint, computational characteristics, machine vector length
Time blocking	Grid size, cache size, footprint, boundary conditions, parallelization
Communication avoiding	Grid size, cache size, footprint, boundary conditions, parallelization
Parallelization	Blockings, machine HW parallelism

Table 8.1: Summary of optimization strategies and what aspects of the computation influences how and if they are applicable to a particular problem.

and the dimensionality. Table 8.1 summarizes the optimizations in this section as well as what aspects of the computation influence their applicability.

8.4 Modeling Performance of Structured Grid Algorithms

Structured grid algorithms have been extensively studied and their performance modeled. In this section, we describe a model for serial memory-bound structured grid algorithms, then describe further refinements to this model for parallel implementations. Finally, we derive an empirical roofline model for structured grid algorithms, which is applicable to both memory-bound and computation-bound kernels, and is the model used in this work.

8.4.1 Serial Performance Models

In previous work [60], we created a serial performance model for memory-bound structured grid computations. Specific to the Rivera cache blocking optimizations on modern architectures described previously, this memory model hinged on the insight that ensuring prefetching performance as well as reducing capacity misses was an important consideration when optimizing. The predictive model uses a proxy microbenchmark called Stanza Triad, which is similar to the STREAM [76] benchmark. However, Stanza Triad performs triad operations on *stanzas* (consecutive memory locations of fixed size) and then jumps forward in memory; this access pattern mimics how blocking changes the addresses the memory hierarchy sees. Modeling performance of Stanza Triad requires a three-point model for performance for the first cache line in a stanza, the second line, and the rest of the data in the stanza, in increasing performance order. Figure 8.12 shows the modeled and actual performance of the microbenchmark on three different architectures.

Based on this model of prefetcher performance, we construct a model for the performance of a blocked structured grid kernel using the same three-point regime and taking into account how many of each type of access occurs in the blocked algorithm to obtain upper and lower bounds on performance. Figure 8.13 demonstrates our model for a heat equation solver on a 512^3 grid on

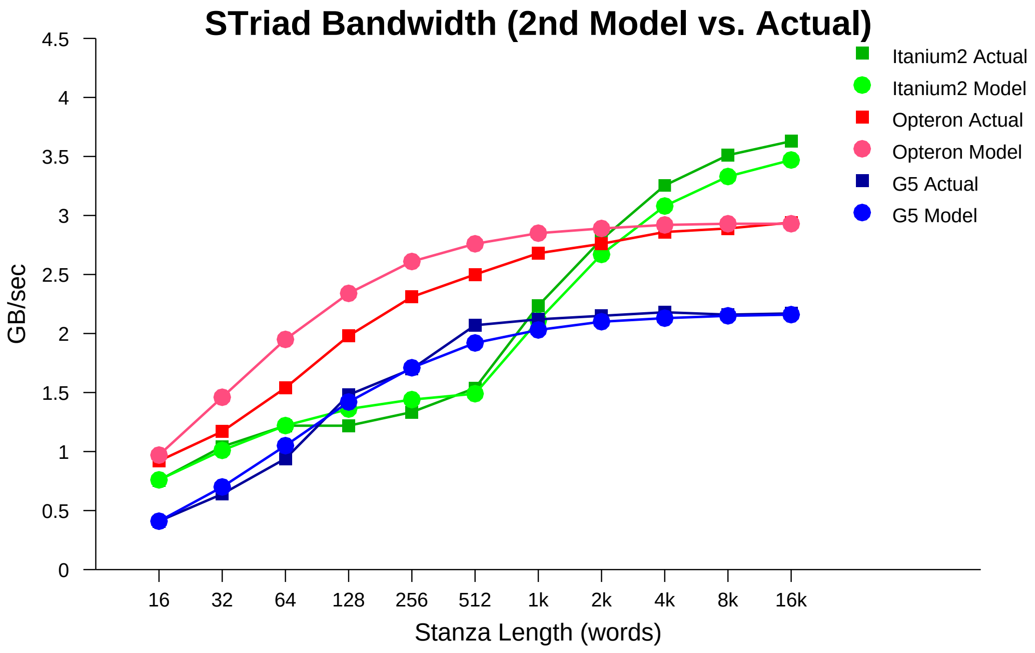


Figure 8.12: Stanza Triad performance and model for three architectures. The benchmark is similar to STREAM, but does "stanzas" of triads before jumping to a new location in memory. This mimics access patterns in structured grid kernels. To model obtained memory bandwidth, we use a three parameter model that separates the first cache line of a stanza, the second, and the rest of the cache lines into three different performance regimes, due to prefetcher performance. Observed and modeled data match well.

a single-core AMD Opteron machine that predicts performance well. In addition, we can apply the model to determine when blocking optimizations will help for the serial algorithm, based on cache size, grid dimensions, and grid dimensionality. Figure 8.14 shows our model for 2D and 3D computations.

In more recent studies, others have created parallel blocking performance models based on this work [30, 29]. Although these are mixed for how well they predict actual performance, they can provide insight into areas of the parameter space for blocking that are unlikely to yield good performance as well as outlining parameter regions likely to yield high performance. Such insights can be useful for auto-tuning.

8.4.2 Roofline Model for Structured Grid

Figure 8.15 shows an example roofline [119] model of a structured grid calculation. To calculate the empirical roofline performance bounds for structured grid kernels, as usual we first create ceilings corresponding to the machine peak memory bandwidth (usually obtained using the STREAM microbenchmark) and the machine's theoretical performance; for most structured grid algorithms, this is the peak floating point operations per second, although depending on the operation used inside the kernel, it may be integer operations or some other computation.

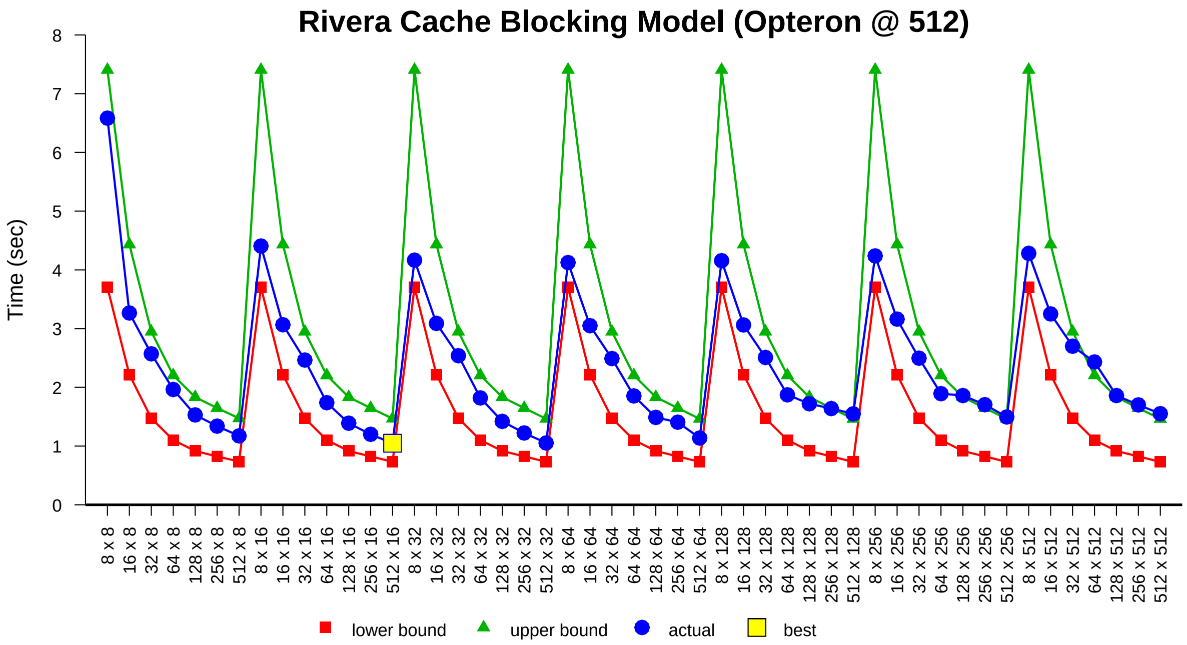


Figure 8.13: Modeled and observed performance for a Laplacian (heat equation) structured grid problem (grid size 512^3) on an AMD Opteron machine, using 2D blocking for optimization. Observed speedups match well our analytic upper and lower bounds based on the Stanza Triad microbenchmark.

We must also create a new ceiling for the bandwidth by crafting a microbenchmark that has the same mix of read and write streams as the kernel under study, since the number and mix of memory streams often changes the obtainable bandwidth due to peculiarities of the memory hierarchy. Furthermore, because structured grid kernels often contain operations that are not straightforward to vectorize, we calculate a new computation ceiling based on in-cache performance of the kernel; this gives a more realistic bound on computation performance given the mix of instructions in the innermost loops. These two ceilings represent tighter limits on obtainable performance for a given structured grid kernel.

Finally, we plot the kernel using a vertical line at the kernel's operational intensity, defined as the ratio of operations to bytes of memory used. This vertical line tells us whether the kernel is bound by memory hierarchy performance or whether it is bound by in-core computation performance. The red and blue dotted lines in Figure 8.15 are examples of memory bandwidth and computation bound kernels, respectively. Note that until operational intensity is high enough, the kernel will be bound by memory bandwidth performance, which is the case for many structured grid kernels.

The models in this section are useful even when employing auto-tuning. In fact, the models themselves may not exactly predict performance, but are better used as tools to understand what kinds of optimizations may improve performance. In the context of auto-tuning, they can also be used for two purposes: first, to determine whether the achieved performance of a particular version is "good enough" relative to the expected peak performance; and second, to help reduce the size of

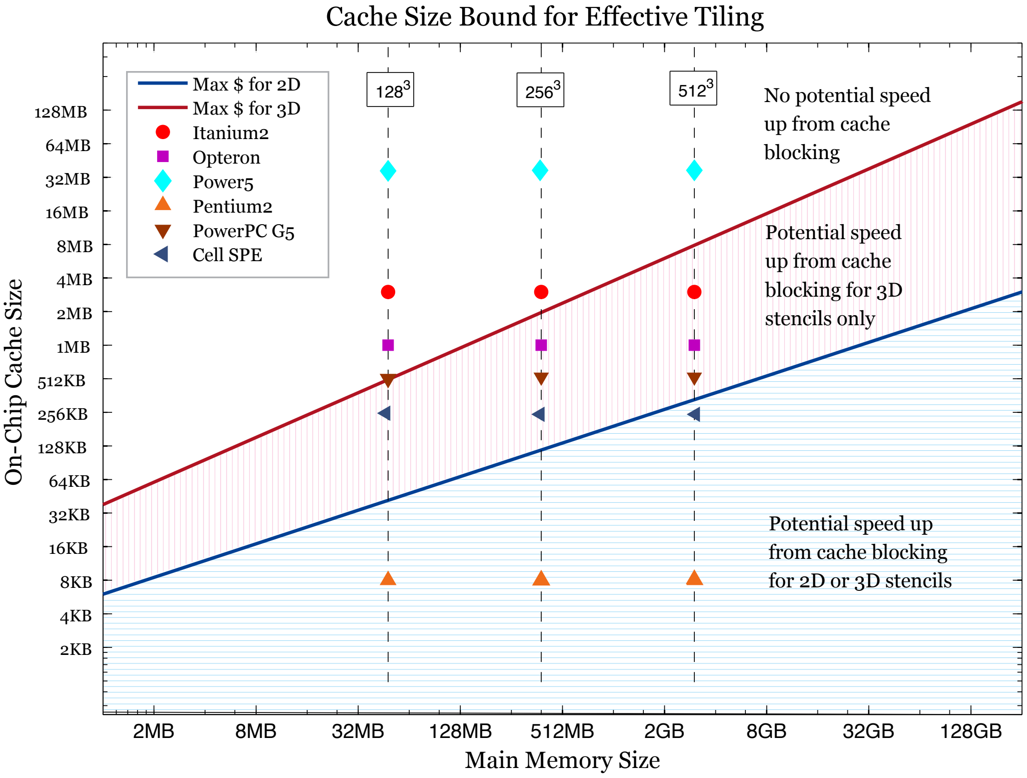


Figure 8.14: Based on cache size and grid dimensions, we can determine whether it will be useful to enable cache blocking, based on our performance model for serial structured grid algorithms.

the search space by eliminating parameter sets that will perform poorly.

8.5 Summary

In this chapter, we explored the various aspects of structured grid computations that make them difficult to encapsulate in a traditional optimized library. Though conceptually simple, computations in this motif vary highly from application to application, and these differences greatly influence the possible optimizations as well as the manner in which these optimizations are applied. In addition, these optimizations influence one another, making traditional code generation approaches such as auto-tuned libraries more difficult. Finally, we defined some metrics to model the bottlenecks of structured grid computations, and outlined how we can use the roofline model to guide our auto-tuning and evaluate its effectiveness.

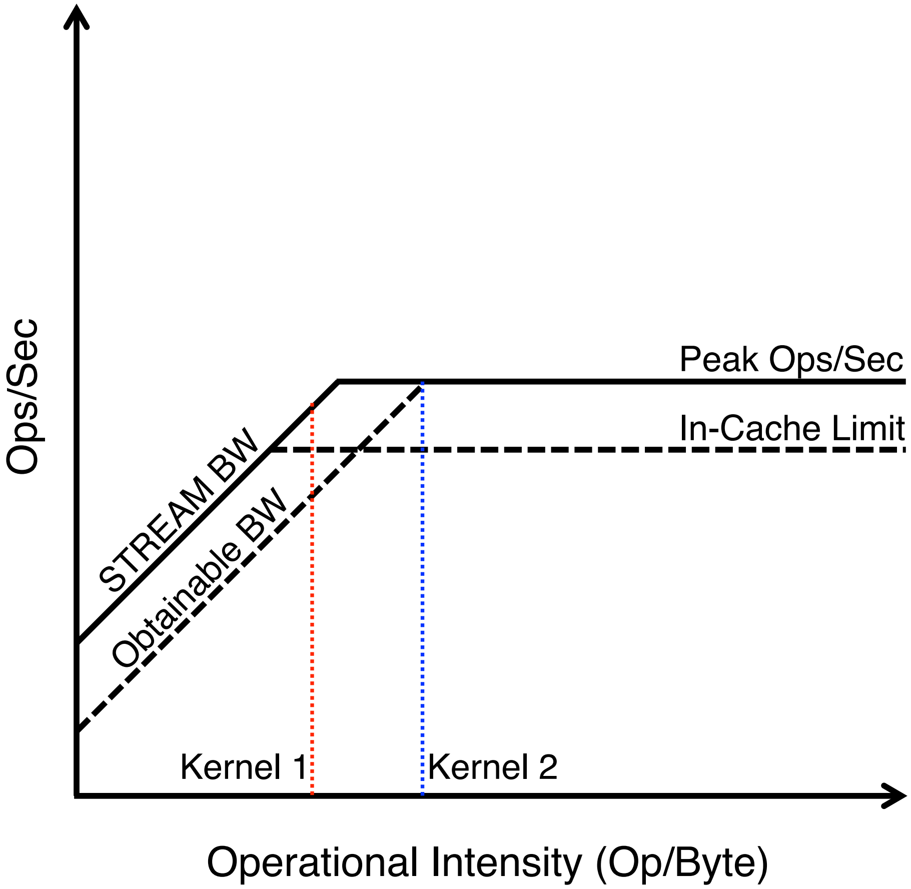


Figure 8.15: Example Roofline model for structured grid calculations. The memory bandwidth and operation rate are plotted using standard microbenchmarks such as STREAM and Linpack. Because different mixes of read/write streams change obtainable bandwidth, further microbenchmarks may be required to find obtainable memory bandwidth. Similarly, the particular mix of operations may result in a lower ceiling than peak operations per second; this bound can be found by running in-cache versions of a kernel. Kernel 1 here is memory bound due to its low operational intensity, while Kernel 2 is bound by computation.

Chapter 9

An Auto-tuner for Parallel Multicore Structured Grid Computations

This chapter describes a proof-of-concept of a generalized auto-tuning approach, which uses a domain-specific transformation and code-generation framework combined with a fully-automated search to replace structured grid kernels written in Fortran with their optimized versions. The interaction with the application program begins with simple annotation of the loops targeted for optimization. The search system then extracts each designated loop and builds a test harness for that particular kernel instantiation; the test harness simply calls the kernel with random data populating the grids and measures performance. Next, the search system uses the transformation and generation framework to apply our suite of auto-tuning optimizations, running the test harness for each candidate implementation to determine optimal performance. After the search is complete, the optimized implementation is built into an application-specific library that is called in place of the original.

The proof-of-concept framework described in this section forms the basis of the embedded DSL in Chapter 10, which expresses more general structured grid computations and is more formally specified, as well as being embedded in a high-level language. In addition, the performance results in Section 9.5 demonstrate the viability of manipulating structured grid code with tree-based optimizations and then auto-tuning over the generated variants to produce high performance implementations. The compiler for the structured grid DSEL in Chapter 10 follows this approach.

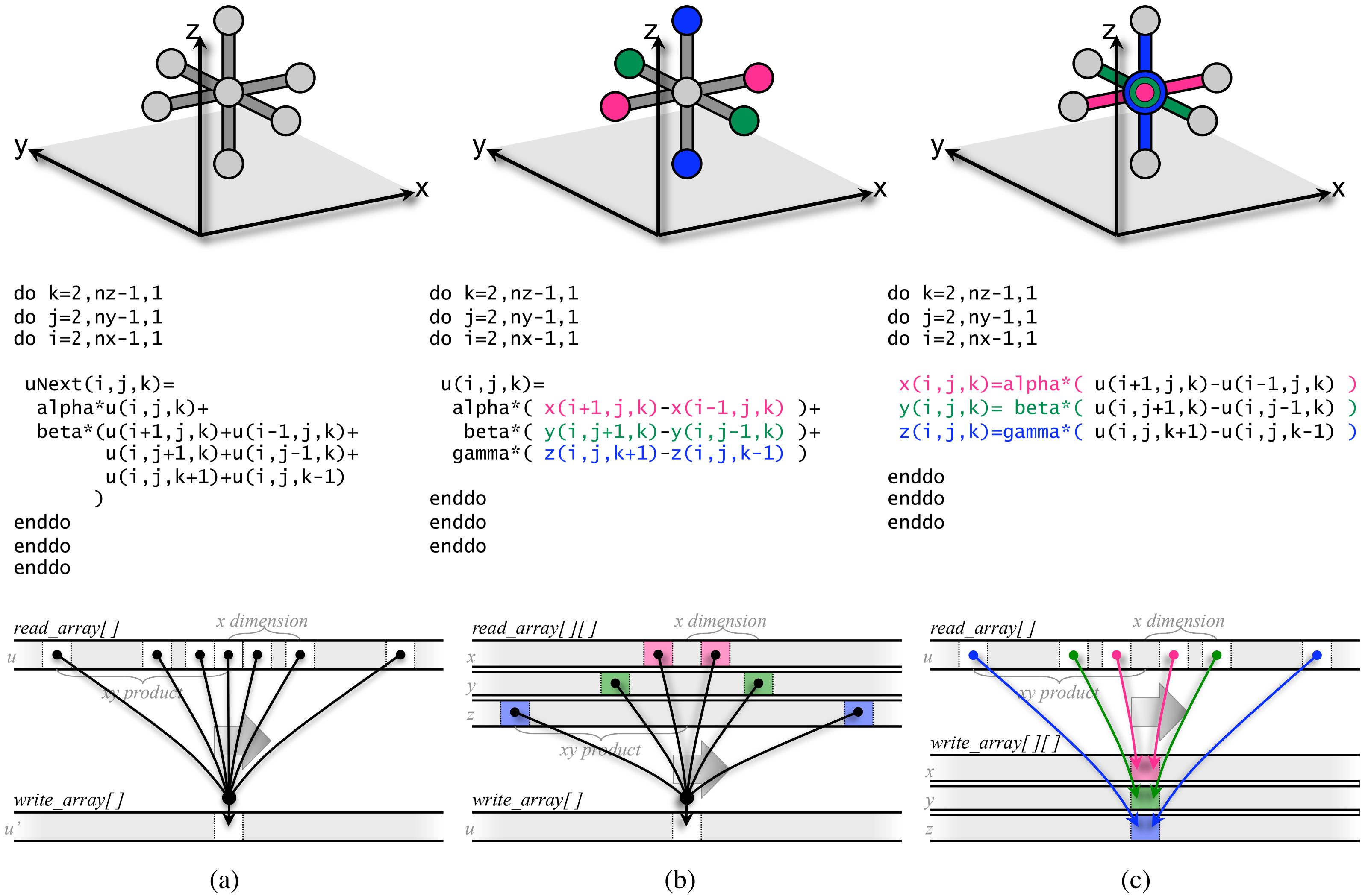
We begin this chapter by describing the architectures and kernels under study in Section 9.1 and the auto-tuning framework in Section 9.2. Optimizations and code generation strategies the auto-tuner uses are described in Sections 9.3 and 9.4. Section 9.5 analyzes the performance obtained by the tuner. Section 9.6 describes some limitations of the proof-of-concept system in this chapter, and Section 9.7 summarizes.

9.1 Structured Grids Kernels & Architectures

This proof-of-concept auto-tuner is used to explore the potential of whether code transformation/generation is a viable path to automatically obtaining high performance for structured grid kernels. Thus, we restrict our tuner to just a few kernels to simplify this exploration. Architectures used in this chapter differ from those in the rest of the thesis, so we will describe them in this

Kernel	Cache References (doubles)	Flops per Stencil	Compulsory Read Traffic	Writeback Traffic	Write Allocate Traffic	Capacity Miss Traffic	Naïve Arithmetic Intensity	Tuned Arithmetic Intensity	Expected Auto-tuning Benefit
Laplacian	8	8	8 Bytes	8 Bytes	8 Bytes	16 Bytes	0.20	0.33	1.66×
Divergence	7	8	24 Bytes	8 Bytes	8 Bytes	16 Bytes	0.14	0.20	1.40×
Gradient	9	6	8 Bytes	24 Bytes	24 Bytes	16 Bytes	0.08	0.11	1.28×

Table 9.1: Structured grid kernel characteristics. Arithmetic Intensity is defined as the Total Flops / Total DRAM bytes. Capacity misses represent a reasonable estimate for cache-based superscalar processors. Auto-tuning benefit is a reasonable estimate based on the improvement in arithmetic intensity assuming a memory bound kernel without conflict misses.



```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 uNext(i,j,k)=
  alpha*u(i,j,k)+
  beta*(u(i+1,j,k)+u(i-1,j,k)+
        u(i,j+1,k)+u(i,j-1,k)+
        u(i,j,k+1)+u(i,j,k-1)
       )
enddo
enddo
enddo
```

```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 u(i,j,k)=
  alpha*( x(i+1,j,k)-x(i-1,j,k) )+
   beta*( y(i,j+1,k)-y(i,j-1,k) )+
  gamma*( z(i,j,k+1)-z(i,j,k-1) )
enddo
enddo
enddo
```

```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 x(i,j,k)=alpha*( u(i+1,j,k)-u(i-1,j,k) )
 y(i,j,k)= beta*( u(i,j+1,k)-u(i,j-1,k) )
 z(i,j,k)=gamma*( u(i,j,k+1)-u(i,j,k-1) )
enddo
enddo
enddo
```

Figure 9.1: (a) Laplacian, (b) divergence, and (c) gradient kernels. Top: 3D visualization of the nearest neighbor structured grid operator. Middle: code as passed to the parser. Bottom: memory access pattern as the kernel sweeps from left to right. Note: the color represents cartesian components of the vector fields (scalar fields are gray). Figure courtesy of Sam Williams.

Core Architecture	**AMD Barcelona**	**Intel Nehalem**	**Sun Niagara2**	**Nvidia GT200 SM**
Type	superscalar out of order	superscalar out of order	HW multithread dual issue	HW multithread SIMD
Clock (GHz)	2.30	2.66	1.16	1.3
DP GFlop/s	9.2	10.7	1.16	2.6
Local-Store	—	—	—	16KB**
L1 Data Cache	64KB	32KB	8KB	—
private L2 cache	512KB	256KB	—	—
System Architecture	**Opteron 2356 (Barcelona)**	**Xeon X5550 (Gainestown)**	**UltraSparc T5140 (Victoria Falls)**	**GeForce GTX280**
# Sockets	2	2	2	1
Cores per Socket	4	4	8	30
Threads per Socket‡	4	8	64	240
primary memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading	Multithreading with coalescing
shared L3 cache	2×2MB (shared by 4 cores)	2×8MB (shared by 4 cores)	2×4MB (shared by 8 cores)	—
DRAM Capacity	16GB	12GB	32GB	1GB (device) 4GB (host)

DRAM Pin
Bandwidth (GB/s)
	21.33	51.2	42.66(read)
			21.33(write)
				141 (device)
				4 (PCIe)

DP GFlop/s	73.6	85.3	18.7	78
DP Flop:Byte Ratio	3.45	1.66	0.29	0.55
Threading	Pthreads	Pthreads	Pthreads	CUDA 2.0
Compiler	gcc 4.1.2	gcc 4.3.2	gcc 4.2.0	nvcc 0.2.1221

Table 9.2: Architectural summary of evaluated platforms. ‡A *CUDA thread block* is considered 1 thread, and 8 may execute concurrently on a streaming multiprocessor. **16 KB local-store shared by all concurrent *CUDA thread blocks* on the streaming multiprocessor.

section.

9.1.1 Benchmark Kernels

To show the broad utility of our framework, we select three conceptually easy-to-understand, yet deceptively difficult-to-optimize structured grid kernels arising from the application of the finite difference method to the Laplacian ($u_{next} \leftarrow \nabla^2 u$), divergence ($u \leftarrow \nabla \cdot \mathbf{F}$) and gradient ($\mathbf{F} \leftarrow \nabla u$) differential operators. Details of these kernels are shown in Figure 9.1 and Table 9.1. All three operators are implemented using central-difference on a 3D rectahedral block-structured grid via Jacobi's method (out-of-place), and benchmarked on a 256^3 grid (not including ghost zones). The Laplacian operator uses a single input and a single output grid, while the divergence operator utilizes multiple input grids and the gradient operator uses multiple output grids; for the latter two, the multiple grid inputs or outputs represent a *structure of arrays* data structure, where the grid contains 3D vectors. In a structure of arrays, each point in the grid has vector data, but this vector data is stored as a separate grid for each scalar in the vector. Although the code generator has no restrictions on data structure, for brevity, we only explore the use of a structure of arrays for vector fields.

Table 9.1 presents the characteristics of the three structured grid operators and sets performance expectations. Like the 3C's cache model [50], we break memory traffic into compulsory read, write back, write allocate, and capacity misses. A naïve implementation will produce memory traffic equal to the sum of these components, and will therefore result in the shown arithmetic intensity ($\frac{totalflops}{totalbytes}$), ranging from 0.20–0.08. As a result, the kernels are bound by memory bandwidth performance on most architectures. The auto-tuning framework in this chapter attempts to improve performance by eliminating capacity misses; thus it is possible to bound the resultant arithmetic intensity based only on compulsory read, write back, and write allocate memory traffic. For the three examined kernels, capacity misses account for dramatically different fractions of the total memory traffic. Thus, we can also bound the resultant potential performance boost from auto-tuning per kernel — 1.66×, 1.40×, and 1.28× for the Laplacian, divergence, and gradient respectively. Moreover, note that the kernel's auto-tuned arithmetic intensity will vary substantially from each other, ranging from 0.33–0.11. Therefore, performance is expected to vary proportionally, as predicted by the Roofline model [119].

9.1.2 Experimental Platforms

To evaluate our structured grid auto-tuning framework, we examine a broad range of leading multicore designs: AMD Barcelona, Intel Nehalem, Sun Victoria Falls, and Nvidia GTX 280. A summary of key architectural features of the evaluated systems appears in Table 9.2; space limitations restrict detailed descriptions of the systems. As all architectures have Flop:DRAM byte ratios significantly greater than the arithmetic intensities described in Section 9.1.1, we expect all architectures to be memory bound. Although the node architectures are diverse, they represent potential building-blocks of current and future ultra-scale supercomputing systems.

9.2 Auto-tuning Framework

Structured grid applications use a wide variety of data structures in their implementations, representing grids of multiple dimensionalities and topologies. Furthermore, the details of the underlying applications call for a variety of numerical kernel operations. Thus, building a static auto-tuning library in the spirit of ATLAS [118] or OSKI [115] to implement the many different structured grid kernels is infeasible.

The overall flow through the auto-tuning system is shown in Figure 9.2. The system begins by parsing user-annotated Fortran code into an Abstract Syntax Tree (AST). Architecture-specific strategy engines then use transformations to manipulate the code into many ASTs that are suitable for code generation. Then, backend-specific code generators create a test harness and many candidate source implementations of the kernel, which are then all run using a search engine that executes each version, finally outputting the best-performing source code for use within an application.

In the next sections, we describe the stages of our auto-tuner flow in more detail.

9.2.1 Front-End Parsing

The front-end for the proof-of-concept tuner parses a description of the structured grid kernel in a subset of Fortran 95; this subset of an existing well-known language was chosen to simplify

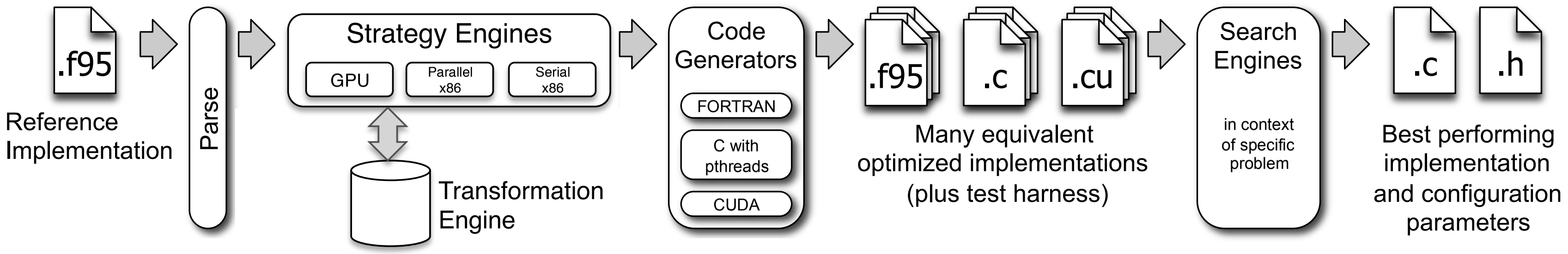


Figure 9.2: Structured grid auto-tuning framework flow. The tuner parses simple domain-specific code into an abstract representation, applies transformations, generates code for specific target backends, and then determines the optimal auto-tuned implementation via search.

the front-end parser and because the primary purpose of the proof-of-concept is to explore the transformations and code generation. Due to the modularity of the transformation engine, a variety of front-end implementations are possible. The result of parsing in our preliminary implementation is an Abstract Syntax Tree (AST) representation of the structured grid kernel, on which subsequent transformations are performed.

9.2.2 Structured Grid Kernel Breadth

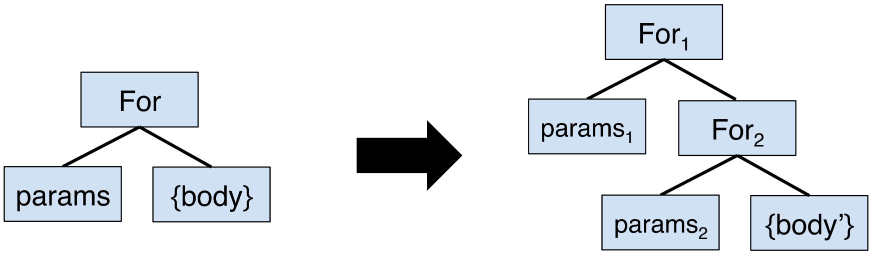
Currently, the auto-tuning system handles a specific class of stencil kernels of certain dimensionality and code structure. In particular, the auto-tuning system assumes a 2D or 3D rectahedral grid, and a kernel based on arithmetic operations and array accesses, with Jacobi-like structure (separate input and output grids). Although this proof-of-concept framework does auto-tune serial kernels with imperfect loop nests, the parallel tuning relies on perfect nesting in order to determine legal domain decompositions and NUMA (non-uniform memory access) page mapping initialization. Additionally, we currently treat boundary calculations as a separate computation.Finally, note that no effort is made to validate that a user-annotated loop actually fits into the class of kernels that our framework can optimize; this shortcoming will be addressed in the next chapter. Overall, our auto-tuning system can target and accelerate a large group of structured grid kernels currently in use, and can be extended to support other kernels in the future.

9.3 Optimization & Code Generation

The heart of the auto-tuning framework is the transformation engine and the backend code generation for both serial and parallel implementations. The transformation engine is in many respects similar to a source-to-source translator, but it exploits domain-specific knowledge of the problem space to implement transformations that would otherwise be difficult to implement as a fully generalized loop optimization within a conventional compiler. Serial backend targets generate portable C and Fortran code, while parallel targets include `pthreads` C code designed to run on a variety of cache-based multicore processor nodes as well as CUDA versions specifically for the massively parallel Nvidia GPUs.

Once the intermediate form is created from the front-end description, it is manipulated by the transformation engine across our spectrum of auto-tuned optimizations. The intermediate form and transformations are expressed in Common Lisp using the portable and lightweight ECL compiler [35], making it simple to interface with the parsing front-ends (written in Flex and YACC) and preserving portability across a wide variety of architectures. Potential alternatives include implemention of affine scaling transformations or more complex AST representations, such as the one used by LLVM [70], or more sophisticated transformation engines such as the one provided by the Sketch [103] compiler.

Because optimizations are expressed as transformations on the AST, it is possible to combine them in ways that would otherwise be difficult using simple string substitution. For example, it is straightforward to apply register blocking either before or after cache-blocking the loop, allowing for a comprehensive exploration of optimization configurations. An example of an AST transformation is shown in Figure 9.3, which shows how a tree is manipulated to perform loop



```
for (int i=0; i<8; i++) {
  body
}
```

```
for (int ii=0; ii<8; ii+=2) {
  for (int i=ii; i<ii+2; i++) {
    body
  }
}
```

Figure 9.3: Example of an AST transformation implementing loop blocking. The tree is manipulated to replace a single loop with two loops, modifying the parameters for the loops to preserve correctness. Bottom shows a pseudocode representation of the results of the transformation.

blocking. In the rest of this section, we discuss serial transformations and code generation, followed by auto-parallelization and parallel-specific transformations and generators.

9.3.1 Serial Optimizations

Several common optimizations have been implemented in the framework as AST transformations, including loop unrolling/register blocking (to improve innermost loop efficiency), cache blocking (to expose temporal locality and increase cache reuse), and arithmetic simplification/constant propagation. These optimizations are implemented to take advantage of the specific domain of interest: Jacobi-like structured grid kernels of arbitrary dimensionality. Possible additional transformations include those shown in previous work [32]: better utilization of SIMD instructions and common subexpression elimination (to improve arithmetic efficiency), cache bypass (to eliminate cache fills), and explicit software prefetching.

A domain-specific code generator run by the user has the freedom to implement transformations that a compiler may not. Although the current set of optimizations may seem identical to existing compiler optimizations, strategies such as memory structure transformations are generally not applied by compilers, since such optimizations are specific to structured grid-based computations. Indeed, we use multidimensional array padding in ways that may be difficult for general compilers to implement automatically. Our restricted domain allows us to make certain assumptions about aliasing and dependencies. Our framework's transformations yield code that outperforms compiler-only optimized versions mostly because compiler algorithms cannot always prove that these (safe) optimizations are allowed.

Given the structured grid transformation framework, we now present parallelization optimizations, as well as cache- and GPU-specific optimizations. The shared-memory parallel code generators leverage the serial code generation routines to produce the version run by each individual thread. Because the parallelization mechanisms are specific to each architecture, both the strategy engines and code generators must be tailored to the desired targets. For the cache-based systems

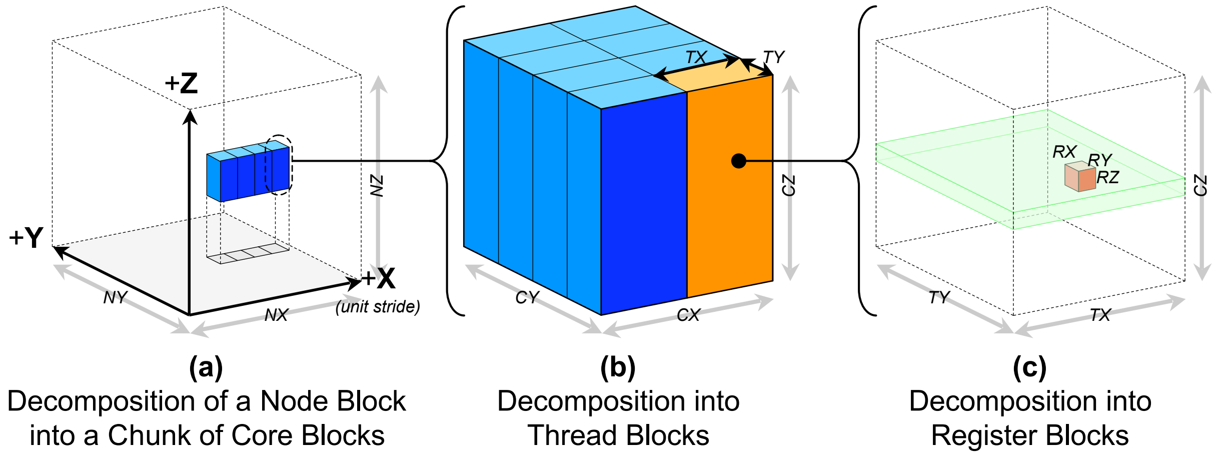

+Z
+Y
+X
(unit stride)
NY
NX
NZ
TX
TY
CZ
CY
CX
RX
RY
RZ
CZ
TY
TX
(a)
Decomposition of a Node Block into a Chunk of Core Blocks
(b)
Decomposition into Thread Blocks
(c)
Decomposition into Register Blocks

Figure 9.4: Four-level problem decomposition: In (a), a *node block* (the full grid) is broken into smaller *chunks*. All *core blocks* in a chunk are processed by the same subset of threads. One core block from the chunk in (a) is magnified in (b). A single *thread block* from the core block in (b) is then magnified in (c). A thread block should exploit common resources among threads. Finally, the magnified thread block in (c) is decomposed into *register blocks*, which exploit data level parallelism.

(Intel, AMD, Sun) we use `pthreads` for lightweight parallelization; on the Nvidia GPU, the only parallelization option is Nvidia's CUDA language and compiler.

Since the parallelization strategy influences code structure, the AST— which represents code run on each individual thread— must be modified to reflect the chosen parallelization strategy. The parallel code generators make the necessary modifications to the AST before passing it to the serial code generator.

9.3.2 Multicore-specific Optimizations and Code Generation

Following the effective blocking strategy presented in previous studies [32], we decompose the problem space into *core blocks*, as shown in Figure 9.4. The size of these core blocks can be tuned to avoid capacity misses in the last level cache. Each core block is further divided into *thread blocks* such that threads sharing a common cache can cooperate on a core block. Though our code generator is capable of using variable-sized thread blocks, we set the size of the thread blocks equal to the size of the core blocks to help reduce the size of the auto-tuning search space. The threads of a thread block are then assigned *chunks* of contiguous core blocks in a round robin fashion until the entire problem space has been accounted for. Finally each thread's structured grid loop is *register blocked* to best utilize registers and functional units. The core block size, thread block size, chunk size, and register block size are all tunable by the framework.

The code generator creates a new set of loops for each thread to iterate over its assigned set of thread blocks. Register blocking is accomplished through strip mining and loop unrolling via the serial code generator.

NUMA-aware memory allocation is implemented by pinning threads to the hardware and taking advantage of first-touch page mapping policy during data initialization. The code generator analyzes the decomposition and has the appropriate processor touch the memory during initialization.

9.3.3 CUDA-specific Optimizations and Code Generation

CUDA [87] programming for Nvidia GPUs is oriented around *CUDA thread blocks*, which differ from the thread blocks used in the previous section. CUDA thread blocks are vector elements mapped to the scalar cores (lanes) of a streaming multiprocessor. The vector conceptualization facilitates debugging of performance issues on GPUs. Moreover, CUDA thread blocks are analogous to threads running SIMD code on superscalar processors. Thus, parallelization on the GTX280, a recent GPU by Nvidia, is a straightforward SPMD domain decomposition among CUDA thread blocks; within each CUDA thread block, work is parallelized in a SIMD manner.

To effectively exploit cache-based systems, code optimizations attempt to employ unit-stride memory access patterns and maintain small cache working sets through cache blocking— thereby leveraging spatial and temporal locality. In contrast, the GPGPU model forces programmers to write a program for each *CUDA thread*. Thus, spatial locality may only be achieved by ensuring that memory accesses of adjacent threads (in a CUDA thread block) reference contiguous segments of memory to exploit hardware coalescing. Consequently, our GPU implementation ensures spatial locality for each point in the structured grid by tasking adjacent threads of a CUDA thread block to perform kernel operations on adjacent grid locations. Some performance will be lost as not all coalesced memory references are aligned to 128-byte boundaries.

The CUDA code generator is capable of exploring the numerous different ways of dividing the problem among CUDA thread blocks, as well as tuning both the number of threads in a CUDA thread block and the access pattern of the threads. For example, in a single time step, a CUDA thread block of 256 CUDA threads may access a tile of $32 \times 4 \times 2$ contiguous data elements; the thread block would then iterate this tile shape over its assigned core block. In many ways, this exploration is analogous to register blocking within each core block on cache-based architectures.

Our code generator currently only supports the use of global “device” memory, and so does not take advantage of the low-latency local-store style “shared” memory present on the GPU. Thus, the generated code does not take advantage of the temporal locality of memory accesses that the use of GPU shared memory provides. The code generator could be modified to generate shared memory code for GPUs.

9.4 Auto-Tuning Strategy Engine

In this section, we describe how the auto-tuner searches the enormous parameter space of serial and parallel optimizations described in previous sections. Because the combined parameter space of the preceding optimizations is so large, it is infeasible to try all possible strategies. In order to reduce the number of code instantiations the auto-tuner must compile and evaluate, we use *strategy engines* to enumerate an appropriate subset of the parameter space for each platform.

The strategy engines enumerate only those parameter combinations (strategies) in the subregion of the full search space that best utilize the underlying architecture. For example, cache blocking in

	Optimization		Parameter Tuning Range by Architecture		
Category	Parameter	Name	Barcelona/Nehalem	Victoria Falls	GTX280
Data Allocation	NUMA Aware		√	√	N/A
Domain Decomposition	Core Block Size	*CX*	NX	{8...NX}	$\{16^\dagger..NX\}$
		CY	{8...NY}	{8...NY}	$\{16^\dagger..NY\}$
		CZ	{128...NZ}	{128...NZ}	$\{16^\dagger..NZ\}$
	Thread Block Size	*TX*	CX	CX	$\{1..\frac{CX}{16}\}^\ddagger$
		TY	CY	CY	$\{\frac{CY}{16}..CY\}^\ddagger$
		TZ	CZ	CZ	$\{\frac{CZ}{16}..CZ\}^\ddagger$
	Chunk Size		$\{1...\frac{NX\times NY\times NZ}{CX\times CY\times CZ\times NThreads}\}$		N/A
Low Level	Array Indexing		√	√	√
	Register Block Size	*RX*	{1...8}	{1...8}	1
		RY	{1...2}	{1...2}	1^*
		RZ	{1...2}	{1...2}	1^*

Table 9.3: Attempted optimizations and the associated parameter spaces explored by the auto-tuner for a 256^3 structured grid problem ($NX, NY, NZ = 256$). All numbers are in terms of doubles. † Actual values for minimum core block dimensions for GTX280 dependent on problem size. ‡ Thread block size constrained by a maximum of 256 threads in a CUDA thread block with at least 16 threads coalescing memory accesses in the unit-stride dimension. *The CUDA code generator is capable of register blocking the Y and Z dimensions, but due to a confirmed bug in the Nvidia nvcc compiler, register blocking was not explored in our auto-tuned results.

the unit stride dimension could be practical on the Victoria Falls architecture, while on Barcelona or Nehalem, the presence of hardware prefetchers makes such a transformation non-beneficial [31].

Further, the strategy engines keep track of parameter interactions to ensure that only legal strategies are enumerated. For example, since the parallel decomposition changes the size and shape of the data block assigned to each thread, the space of legal serial optimization parameters depends on the values of the parallel parameters. The strategy engines ensure all such constraints (in addition to other hardware restrictions such as maximum number of threads per processor) are satisfied during enumeration.

For each parameter combination enumerated by the strategy engine, the auto-tuner's search engine then directs the parallel and serial code generator components to produce the code instantiation corresponding to that strategy. The auto-tuner runs each instantiation and records the time taken on the target machine. After all enumerated strategies have been timed, the fastest parameter combination is reported to the user, who can then link the optimized version of the structured grid kernel into their existing code.

Table 9.3 shows the attempted optimizations and the associated parameter subspace explored by the strategy engines corresponding to each of our tested platforms. While the search engine currently does an exhaustive search over the parameter subspace dictated by the strategy engine (which is a subset of the entire possible optimization space), future work could include more intelligent search mechanisms such as hill-climbing or machine learning techniques [41], where the search engine can use timing feedback to dynamically direct the search.

9.5 Performance Evaluation

In this section, we examine the performance quality and expectations of our auto-parallelizing and auto-tuning framework across the four evaluated architectural platforms. The impact of our framework on each of the three kernels is compared in Figures 9.5–9.7, showing performance of the original serial kernel (gray), auto-parallelization (blue), auto-parallelization with NUMA-aware initialization (purple), and auto-tuning (red). The GTX280 reference performance (blue) is based on a straightforward implementation that maximizes CUDA thread parallelism. We do not consider the impact of host transfer overhead; previous work [32] examined this potentially significant bottleneck in detail. Overall, results are ordered such that threads first exploit multithreading within a core, then multiple cores on a socket, and finally multiple sockets. Thus, on Nehalem, the two thread case represents one fully-packed core; similarly, the GTX280 requires at least 30 CUDA thread blocks to utilize the 30 cores (streaming multiprocessors).

9.5.1 Auto-Parallelization Performance

The auto-parallelization scheme specifies a straightforward domain decomposition over threads in the least unit-stride dimension, with no core, thread, or register blocking. To examine the quality of the framework's auto-parallelization capabilities, we compare performance with a parallelized version using OpenMP [89], along with memory allocation that ensures proper NUMA-aware memory decomposition via first-touch pinning policy. Results, shown as yellow diamonds in Figures 9.5–9.7, show that performance is well correlated with our framework's NUMA-aware auto-parallelization. Furthermore, our approach slightly improves Barcelona's performance, while Nehalem and Victoria Falls see up to a 17% and 25% speedup (respectively) compared to the OpenMP version, indicating the effectiveness of our auto-parallelization methodology even before auto-tuning.

9.5.2 Performance Expectations

When tuning any application, it is important to know when one has reached architectural peak performance, and have little to gain from continued optimization. We make use of a simple empirical model based on the Roofline performance model to establish this point of diminishing returns and use it to evaluate how close our automated approach can come to machine limits. We now examine achieved performance in the context of this simple model based on hardware characteristics. Assuming all kernels are memory bound and do not suffer from an abundance of capacity misses, we approximate the performance bound as the product of streaming bandwidth and each structured grid kernel's arithmetic intensity (0.33, 0.20 and 0.11 — as shown in Table 9.1). Using an optimized version of the STREAM benchmark [30], which we modified to reflect the number of read and write streams for each kernel, we obtain expected peak performance based on memory bandwidth for the CPUs. For the GPU, we use two versions of STREAM: one that consists of exclusively read traffic, and another that is half read and half write.

Our model's expected performance range is represented as a green line (for the CPUs) and a green region (for the GPUs) in Figures 9.5–9.7. For Barcelona and Nehalem, our optimized kernels obtain performance essentially equivalent to peak memory bandwidth. For Victoria Falls, the obtained bandwidth is around 20% less than peak for each of the kernels, because our framework does not

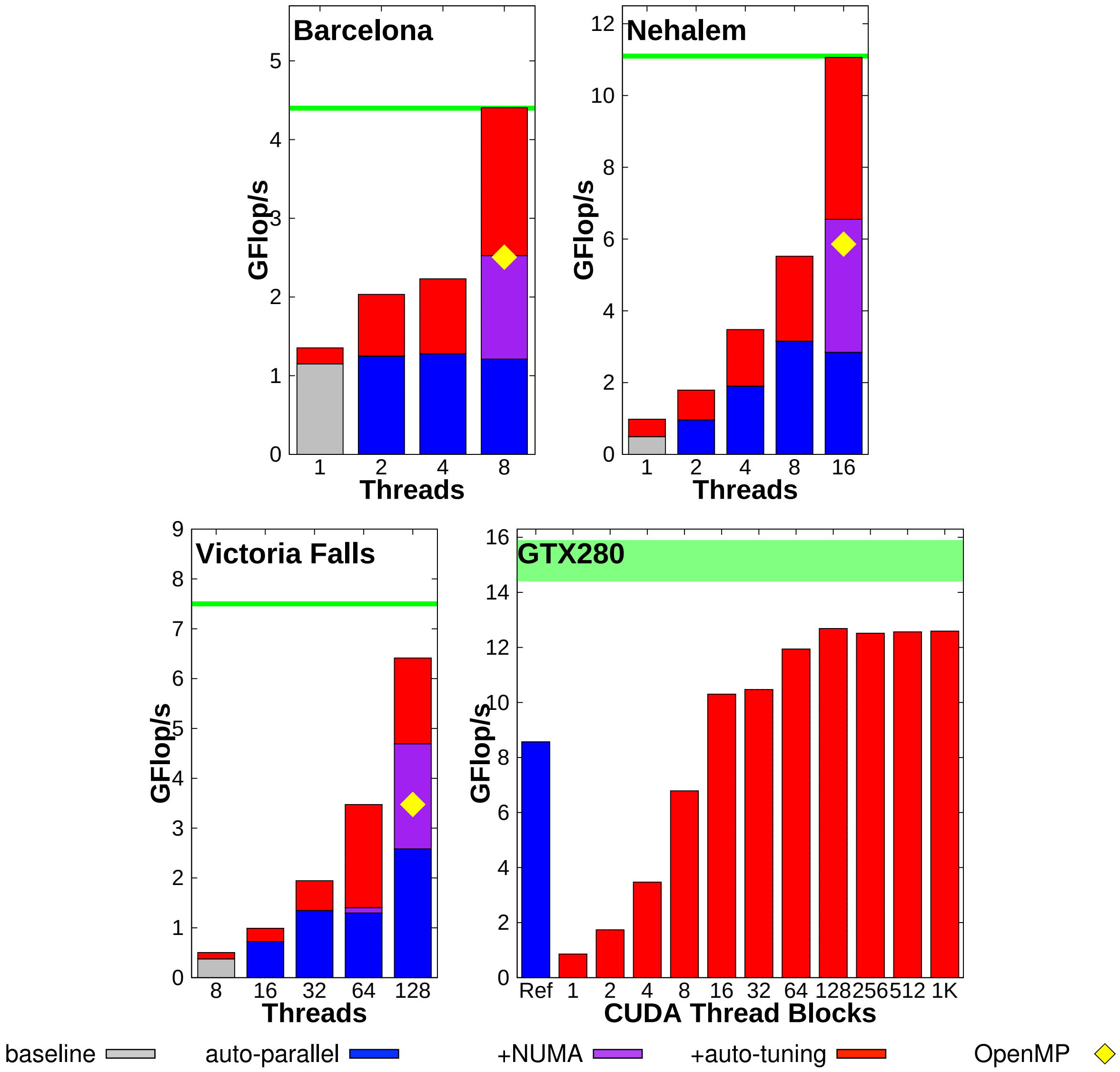


Figure 9.5: Laplacian performance as a function of auto-parallelization and auto-tuning on the four evaluated platforms. The green region marks performance extrapolated from STREAM bandwidth. For comparison, the yellow diamond shows performance achieved using the original structured grid kernel with OpenMP pragmas and NUMA-aware initialization.

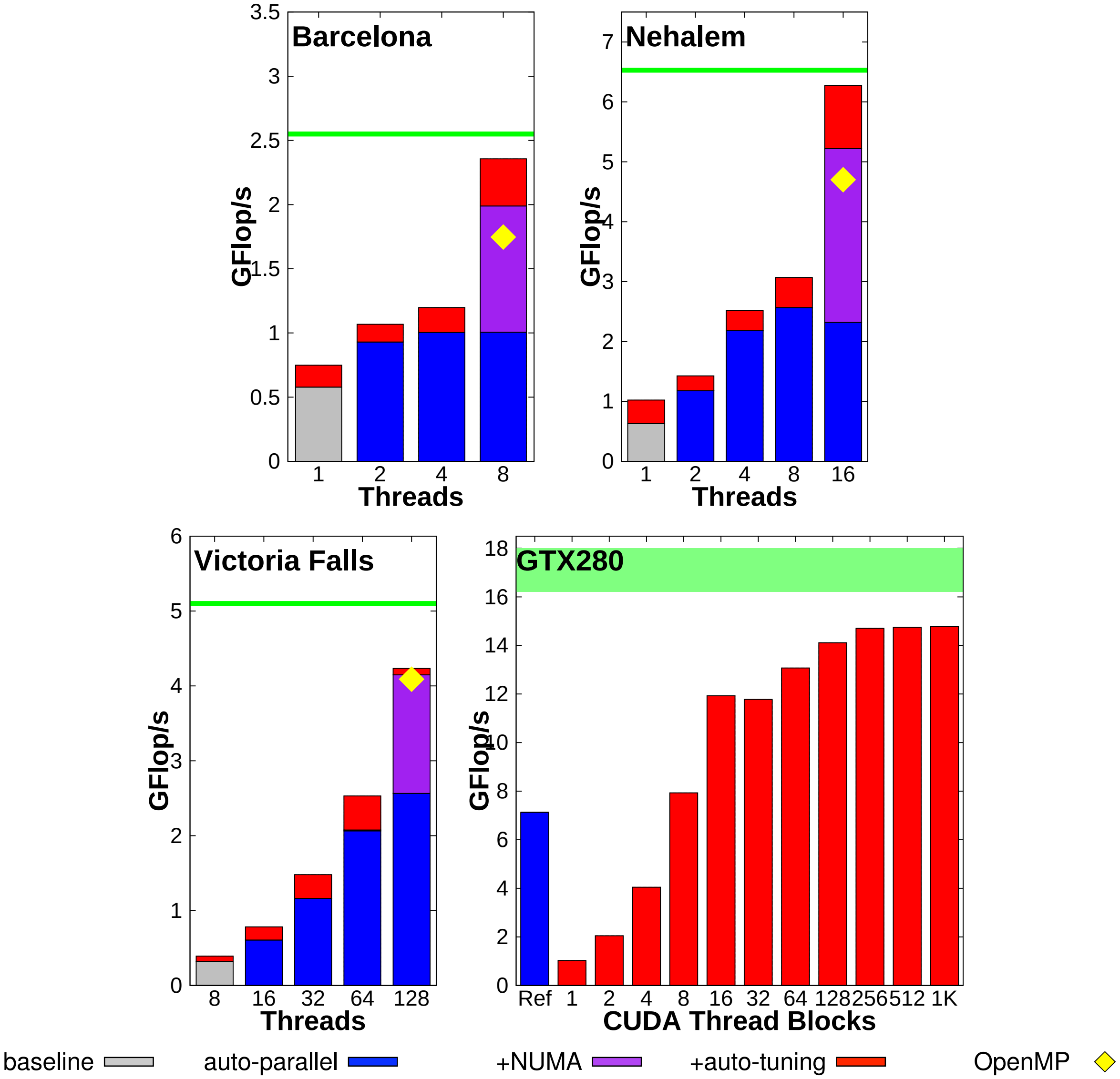


Figure 9.6: Divergence performance as a function of auto-parallelization and auto-tuning on the four evaluated platforms. The green region marks performance extrapolated from STREAM bandwidth. For comparison, the yellow diamond shows performance achieved using the original structured grid kernel with OpenMP pragmas and NUMA-aware initialization.

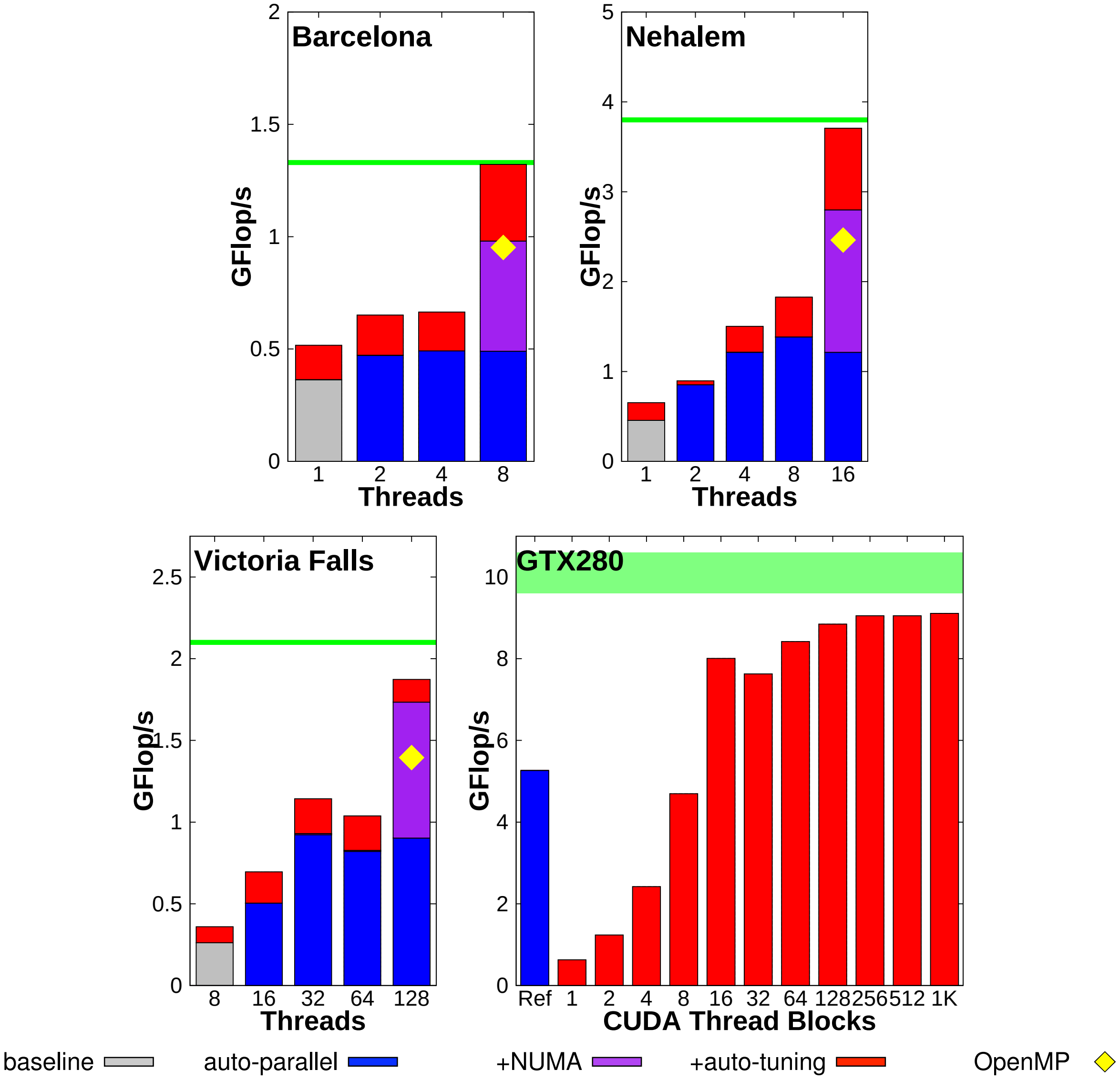


Figure 9.7: Gradient performance as a function of auto-parallelization and auto-tuning on the four evaluated platforms. The green region marks performance extrapolated from STREAM bandwidth. For comparison, the yellow diamond shows performance achieved using the original structured grid kernel with OpenMP pragmas and NUMA-aware initialization.

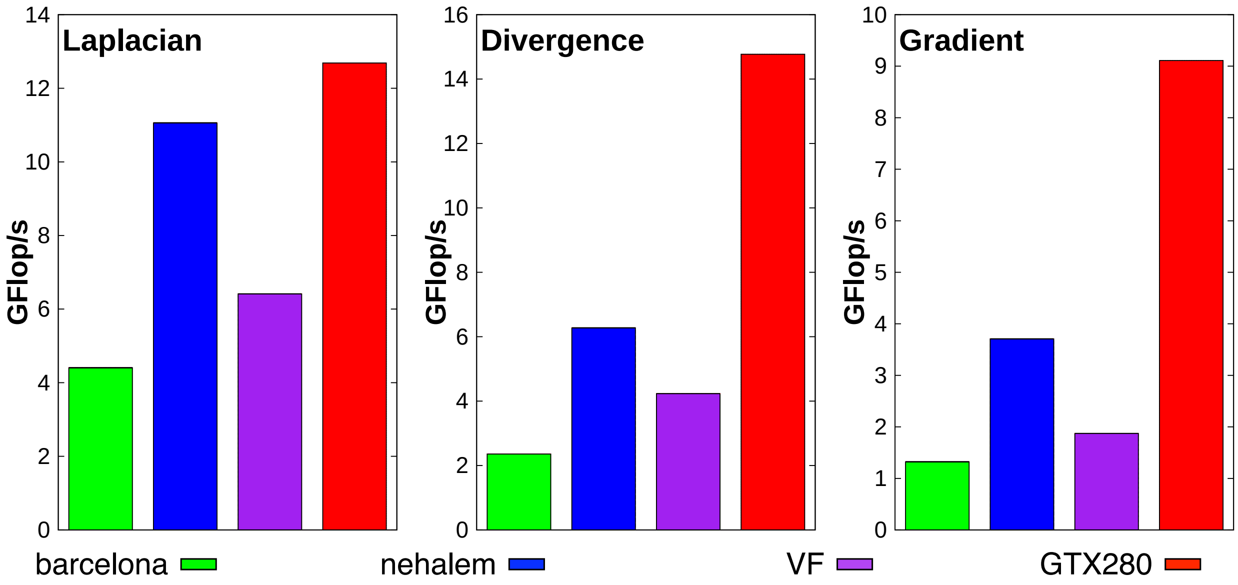


Figure 9.8: Peak performance after auto-tuning and parallelization.

currently implement software prefetching and array padding, which are critical for performance on this architecture. Finally, the GTX280 results are also below our performance model bound, likely due to no array padding [32]. Overall, our fully-tuned performance closely matches our model's expectations, while highlighting areas that could benefit from additional optimizations.

9.5.3 Performance Portability

The auto-tuning framework takes a serial specification of the structured grid kernel and achieves a substantial performance improvement, due to both auto-parallelization and auto-tuning. Overall, we achieve substantial performance improvements across a diversity of architectures— from GPUs to multi-socket multicore x86 systems. Barcelona and Nehalem see between $1.7\times$ and $4\times$ improvement for both the one and two socket cases over the conventional parallelized case, and up to $10\times$ improvement over the serial code. The results also show that auto-tuning is essential on Victoria Falls, enabling much better scalability and increasing performance by $2.5\times$ and $1.4\times$ on 64 and 128 threads respectively in comparison to the conventional parallelized case, but a full $22\times$ improvement over an unparallelized example. Finally, auto-tuning on the GTX280 boosted performance by $1.5\times$ to $2\times$ across the full range of kernels — a substantial improvement over the baseline CUDA code, which is implicitly parallel. This clearly demonstrates the performance portability of this framework across the sample kernels.

The auto-tuner is able to achieve results that are extremely close to the architectural peak performance of the system, which is limited ultimately by memory bandwidth. This level of performance portability using a common specification of kernel requirements is unprecedented for structured grid codes.

9.5.4 Programmer Productivity Benefits

We now compare our framework's performance in the context of programming productivity. Previous work [32] presented the results of Laplacian kernel optimization using a hand-written auto-tuning code generator, which required months of Perl script implementation, and was inherently limited to a single kernel instantiation. In contrast, utilizing our framework across a broad range of possible structured grid kernels only requires a few minutes to annotate a given kernel region, and pass it through our auto-parallelization and auto-tuning infrastructure, thus tremendously improving productivity as well as kernel extensibility.

Currently our framework does not implement several hand-tuned optimizations [32], including SIMDization, padding, or the employment of cache bypass (*movntpd*). However, comparing results over the same set of optimizations, we find that our framework attains excellent performance that is comparable to the hand-written version. We obtain near identical results on the Barcelona and even higher results on the Victoria Falls platform (6 GFlop/s versus 5.3 GFlop/s). A significant disparity is seen on the GTX280, where previous hand-tuned Laplacian results attained 36 GFlop/s, compared with our framework's 13 GFlop/s. For the CUDA implementations, our automated version only utilizes optimizations and code structures applicable to general structured grid kernels, while the hand-tuned version explicitly discovered and exploited the temporal locality specific to the Laplacian kernel— thus maximizing performance, but limiting the method's applicability.

9.5.5 Architectural Comparison

Figure 9.8 shows a comparative summary of the fully-tuned performance on each architecture. The GTX280 consistently attains the highest performance, due to its massive parallelism at high clock rates, but transfer times from system DRAM to board memory through the PCI Express bus are not included and could significantly impact performance [32]. Nehalem obtains the best overall performance of the cache-based systems, due to the combination of high memory bandwidth per socket and hardware multithreading to fully utilize the available bandwidth. Additionally, Victoria Falls obtains high performance, especially given its low clock speed, thanks to massive parallelism combined with an aggregate memory bandwidth of 64 GB/s.

9.6 Limitations

This tuner is able to obtain a large percentage of Roofline performance for at least three different structured grid kernels across a number of architectures. The proof-of-concept successfully demonstrates that domain-specific transformations are capable of obtaining high performance across a number of kernels for the structured grid domain. However, the tuner has a few limitations.

The tuner implemented here operates directly on syntactic tree of the computation and attempts to infer properties of the structured grid from the program text. In addition, there is no check for correctness or analysis to determine whether the computation under study can be transformed properly and still obtain a correct result. In other words, from the perspective of a user, either the structured grid kernel is transformed and obtains correct results, or the kernel is transformed and obtains an incorrect result, or the auto tuning framework crashes. This user experience is less than ideal.

Furthermore, in speaking to domain scientists, the model of using an external auto-tuner that examines the program text is far from the ideal of "Matlab-like" expressiveness and ease-of-use. It is also difficult to integrate with an existing build system and prone to error. Transformations that could improve performance are not just locally-constrained; some require changing the behavior of the entire program. For example, enabling NUMA-aware allocation requires changing the initialization of the data structure, which often occurs far away from the actual calculation. Another example is that ensuring that the data structures are laid out and traversed in the proper order requires making sure the data allocation and traversal both use the same order, for all kernels in an application.

From the perspective of an auto-tuner developer, much of the infrastructure created in order to develop this proof of concept could be abstracted away and be reused for other auto tuning systems. Such a library or framework would make the development of auto-tuning systems that manipulate program text far easier. In addition, although the language used, Lisp, is relatively productive and well-suited for the task of tree transformation, neither the user base of expert auto-tuning system writers nor the user base of application writers are generally familiar with the language.

Finally, the brute force search method employed here may be overkill in many circumstances. The large number of layers of blocking and many other optimizations may be unnecessary. Such a determination requires looking at the data to see which optimizations yield the highest performance, perhaps with decision-tree machine learning approaches to automatically infer the highest-value optimizations and parameters. By analyzing which optimizations were most important for speeding up the kernels, we may be able to create a simpler tuner that has the same level of performance, without the many layers of complexity present in this proof of concept.

These limitations point to new directions in which auto tuning frameworks and systems for other domains can be developed, and motivate the use of the SEJITS methodology for our DSEL in the next chapter.

9.7 Summary

This chapter described a proof-of-concept auto-tuning framework for structured grid computations that used domain-specific AST transformations combined with architecture-specific strategy engines to explore a large space of possible implementations for each kernel. Our approach obtained excellent speedups and high percentages of attainable performance across architectures, but suffered from a number of shortcomings and limitations that motivate our DSEL for this domain. The proof-of-concept was successful, however, in showing that structured grid computations can be optimized in a general and reusable way using domain-specific code transformation and generation.

Chapter 10

Sepya: An Embedded Domain-Specific Auto-tuning Compiler for Structured Grids

This chapter describes Sepya (Stencils Embedded in PYthon with Auto-tuning), an embedded domain-specific language and auto-tuning compiler for structured grid computations. The language, embedded in Python, transforms user-supplied stencil functions into high performance parallel code for multicore and is capable of obtaining a high percentage of peak performance while allowing programmers to express their computations in a high-level language. The DSEL and compiler differ from the previous chapter in the following ways:

- The DSEL we build here supports a larger scope of structured grid computations.
- The language is formally defined, and Sepya ensures the input is validated as a supported structured grid computation before any effort to optimize. This prevents correctness issues that could arise in the original tuner.
- Sepya avoids analysis by validating input code during the conversion to a declarative intermediate form.
- Instead of building up an entire code transformation infrastructure, the Sepya leverages Asp for many operations, reducing the complexity of the DSEL compiler. Furthermore, it is written in Python, a language familiar to a large community of programmers.
- Currently, Sepya only targets multicore CPUs; no GPU code is generated. However, the multicore CPU code is far simpler than the earlier tuner, as the set of possible optimizations is reduced to those that proved to be beneficial in the original proof-of-concept.
- Sepya supports optimizing and generating code for multi-timestep structured grid computations, which were not supported in the earlier proof-of-concept.
- Finally, this DSEL benefits from all the advantages of being embedded and using our framework, including fewer lines of code as well as enabling domain scientists to benefit from high-level languages and existing libraries in them. The resulting code has Matlab-like ease of use.

Section 10.1 describes the implemented language, including the intermediate representation and the language semantics. In Section 10.2, the structure of the Sepya compiler is described. Section 10.3 outlines the implemented algorithms and optimizations. Performance and expressibility are evaluated in Section 10.4, including a brief discussion on productivity. Section 10.5.1 describes several possible extensions to the language, extending its ability to express additional kernels. Finally, Section 10.6 summarizes.

10.1 Analysis-Avoiding DSEL for Structured Grids

This section outlines the design of an embedded DSL for structured grid calculations. Defining the scope of the language requires a balance between two motivating factors. First, the language must be broad enough to support many structured grid kernels; otherwise it is not useful. Secondly, the language must be restricted enough to make constructing an embedded DSL compiler tractable. The smaller the scope, the easier it is to generate correct code that obtains performance, while eliminating the need for complicated analyses.

To explore the minimum scope of a useful structured grid language, we begin by defining a set of building blocks common to many kernels in the motif.

10.1.1 Building Blocks of Structured Grid Calculations

From the characteristics of structured grid computations, as described in Chapter 8, we find that a DSEL for structured grid computations must include the following abstractions:

- A multidimensional data structure for the grid. Grids can be inputs or outputs (for Jacobi-style computations), though some applications use a single grid for both input and output (for Gauss-Seidel or Successive Over Relaxation, which improve convergence properties and reduce memory footprint).

- A control structure that iterates over the grid, differentiating between interior points and border points, since many computations apply separate functions for the two parts of the grid.

- An operator function to be applied at each point in the grid, and, optionally, a separate operator to be applied at each boundary point. These operators can be thought of as pure side-effect-free functions that take as inputs some subset of the neighbors of the central point and output a new value for the center.

- Within the operator, some way to name neighbors of the center point. In many cases, these neighbors are grouped into sets that are treated somewhat differently (for example, different sets may have different weights associated with them).

- In some structured grid computations, an abstraction for timesteps, expressing the repeated application of the operator over the entire grid a fixed number of times. This is common in iterative algorithms, and between timesteps input and output grids are switched.

This is by no means a comprehensive list; additional abstractions and constructs, such as indirect lookups for coefficients, will increase the number of expressible structured grid kernels. However,

Restriction	Analysis Avoided
Input grids and output grids must be disjoint	Aliasing
Number of timesteps must be static	Tailor code generation for number of timesteps
Neighbors must be predefined by user	No need to analyze code for operator footprint, easing all optimizations and preventing users from changing neighbor sets on the fly
Relationship of input/output grids explicit for multiple timesteps	No need for analysis to determine current input and output grids
Limit math functions allowed	Prevent side-effects in external functions
Specify separate kernels for interior and border	Allow separate optimization of interior and border without needing to detect which is which
Only allow weight lookups into a 1D table	Prevents overriding our restriction that neighbors should be the same for each point in the multidimensional grid

Table 10.1: Summary of restrictions for the Sepya DSEL and their benefits.

these constructs are sufficient to cover a large number of widely-used kernels (in fact, a larger space than the proof-of-concept in Chapter 9) and present a minimal set of abstractions necessary to build a useful language for structured grid kernels.

10.1.2 Language and Semantics

In our approach, we define the intermediate representation, called the *Semantic Model* [38]. This Semantic Model will determine the scope of the language, formally defining the kinds of computations the language can express. The goal in defining this intermediate representation is to create a language that implements the abstractions above while minimizing the need for analysis; therefore, it will be necessary to define some restrictions on the various constructs. In addition, the abstractions should have concrete semantics that conform to a domain scientist's conceptual model of structured grids. The Semantic Model is defined similarly to a syntax definition, but encodes the semantics of a calculation as opposed to syntax.

Figure 10.1 shows the Semantic Model for the structured grid DSEL. Note that this language is quite a bit broader than the minimal set of constructs defined earlier; however, the additional constructs have been chosen to add relatively large amounts of expressibility in exchange for the added complexity. An example of such a construct is the `InputElementExprIndex` node, which allows indexing a single-dimensional weight array. This allows expressing additional structured grid kernels, but adds little complexity since the array is generally assumed to be much smaller than the sizes of the input and output arrays.

Some of the operational semantics of the constructs are worth examining in more detail. In the `StencilModel`, there is an `interior_kernel` which is applied to the iteration space of interior points of the input grids in an unordered manner. In other words, the computation is

$$\forall x \in interior(input), outputs(x) = f(x, inputs, outputs)$$

```
# Constraint checks are not shown

# Top-level node for an instance of a semantic model
StencilModel(input_grids=Identifier*, output_grids=Identifier*, interior_kernel=Kernel,
             border_kernel=Kernel)

# a kernel either does one or many timesteps
Kernel(body=TimeStepIter | (StencilNeighborIter | OutputAssignment)*)

# iterator over neighbors
StencilNeighborIter(grid=Identifier, neighbors_id=Constant, body=OutputAssignment*)

# assigns value Expr to current output element
OutputAssignment(target=OutputElement, value=Expr)

# iterates over timesteps
TimeStepIter(limit=types.IntType, grid_rotate=GridRotate*,
             body=(StencilNeighborIter | OutputAssignment)*)

# rotates grids left
GridRotate(grids=Identifier*)

Expr = Constant
     | Neighbor      # Refers to current neighbor inside a StencilNeighborIter
     | OutputElement # Refers to current output element
     | InputElement  # An element of input grid at a neighbor location
     | InputElementZeroOffset # The element of the input grid at the center location
     | InputElementExprIndex  # Element used to lookup into a 1D grid (for coefficients)
     | ScalarBinOp
     | MathFunction

# the building block for the function applied at each point
ScalarBinOp(left=Expr, op=(ast.Add|ast.Sub|ast.Mult|ast.Div|ast.FloorDiv|ast.Mod),
            right=Expr)

# we only allow a few math functions
MathFunction(name, args=Expr*)
```

Figure 10.1: Semantic Model for the structured grid language.

where x is a point in the interior, and the function f is the interior kernel. No ordering can be assumed for x. Similarly, the `StencilNeighborIter` iterates over the points in the numbered set of neighbors (i.e. `neighbors_id`) of the grid, and executes the body. Again, the iteration is unordered.

An `OutputAssignment` node assigns to its `target` (which is one of the interior points of the grid, assuming it is an interior iteration) the expression in the `value` field. Note that the way this is constructed forces the left hand side of the assignment to be into an output grid, and that the way the right hand expression is constructed prevents the right hand side from accessing a neighbor element in the output grid.

A final node to examine is the `GridRotate` construct, which is used to change the assignment of input/output grids between timesteps. For example, in a calculation with a single input and output grid, between timesteps the two grids are almost always switched. The semantics of the construct are that after its execution,

$$grids[i\%L] = grids[(i+1)\%L]$$

where L is the number of grids. In other words, each identifier now points to the grid its successive grid pointed to earlier.

10.1.3 Avoiding Analysis

In order to make it "correct-by-construction" (i.e. analysis-avoiding), we limit the Semantic Model as much as possible while retaining the ability to express kernels. For example, the structure of the Semantic Model does not allow the set of neighbors to be dependent on the location within the grid (other than the distinction between the interior or border). As another example, the kernel cannot change at all based on the timesteps, since the current timestep cannot be named in the kernel. Similarly, the `OutputAssignment` node described above is restricted to prevent expressing calculations that are not structured grid computations. These limitations, expressed through the structure of the intermediate representation, help restrict the complexity of the language and compiler, while preserving the most common patterns used in structured grid computations. A summary of the restrictions encoded in our Semantic Model is shown in Table 10.1, along with the benefits of each restriction.

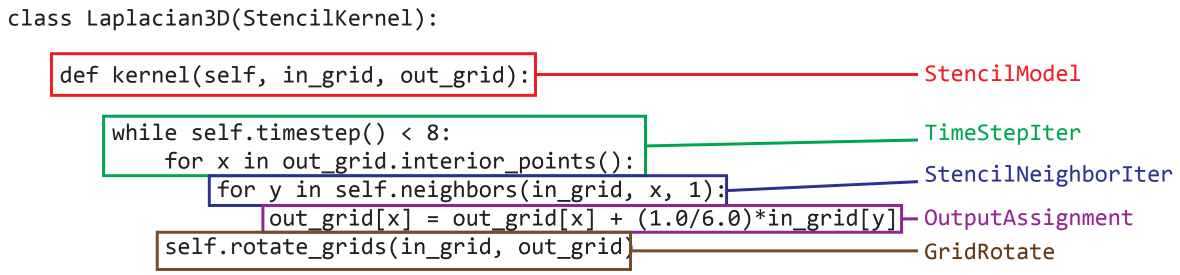
One major restriction is that the sets of neighbors (multiple sets are allowed, and iterators can iterate over a particular set only) are defined outside of the kernel function, and cannot be redefined within the kernel. Although this set could be inferred using relatively simple static analysis, we force the user to pre-declare the neighbors for two reasons. First, this prevents the user from changing the neighbor set from with the kernel. Secondly, this explicit statement of the shape reduces the need for implementing analysis when transforming the loop into blocked or unrolled code, which require the shape of the kernel to be known. Thus, this restriction helps simplify the compiler implementation by forcing the user to convey explicit information to the compiler as well as by ensuring the kernel does not use features unsupported by the compiler.

Having examined the Semantic Model and how it avoids analysis while expressing a large set of structured grid kernels, we now turn to the embedding of this language into Python.

10.1.4 Language in Python Constructs

While the Semantic Model defines the internal representation, and therefore the semantics of what structured grids are expressible, users write their code in the embedding language. We choose Python for our DSEL, though others could be used. We must define a mapping from pure Python syntax into the Semantic Model; this mapping will be used by Sepya to construct instances of the intermediate representation, during which correctness checking will occur.

The primary consideration when designing this mapping is to consider, as much as possible, the semantics of pure Python to design the mapping in a way that looks familiar to the domain scientist, while limiting the complexity for the compiler. In addition, the mapping should be expressed as an API for programmers to use, while specifying any restrictions/assumptions needed by the DSEL



```
class Laplacian3D(StencilKernel):
    def kernel(self, in_grid, out_grid):
        while self.timestep() < 8:
            for x in out_grid.interior_points():
                for y in self.neighbors(in_grid, x, 1):
                    out_grid[x] = out_grid[x] + (1.0/6.0)*in_grid[y]
            self.rotate_grids(in_grid, out_grid)
```

Figure 10.2: Example of a multi-timestep structured grid computation, embedded in Python, showing the correspondence with major nodes in the Semantic Model.

compiler. This eliminates the need for application writers to understand the internals of the Semantic Model.

From the DSEL user's perspective, the language can be summarized with the following pseudo-API:

- A structured grid kernel is encapsulated in a class that inherits from the `StencilKernel` class.

- The actual computation is in an instance method called `kernel()`, with parameters that are the input and output grids for the computation. The kernel function consists of one or more iterators over grid interiors or border points.

- Within an `interior_iterator`, one can iterate over the points that are neighbors to the center point by using the `neighbors` iterator, which yields each neighboring point.

- The order in which interior points and neighbor points are iterated over is not guaranteed.

- The computation at each point has an output grid on the left hand side, and on the right hand side can only access the center point of the output grid, the center point of the input grid(s), or the neighbor point of the input grid(s). A few other kinds of accesses (for example, into a lookup array for coefficients) are also possible.

- All input arrays (except 1D lookup tables) and output arrays must have the same sizes and dimensionality in order for the iterators to work properly. This restriction can be addressed with language extensions described in Section 10.5.1.

This relatively succinct description describes the language in enough detail to enable using the structured grid DSEL. Figure 10.3 shows a full example of defining a structured grid kernel and calling it. Another example, which shows the correspondence between Python syntax and the intermediate representation nodes, is shown in Figure 10.2.

In addition, we ensure that this language defined as pure Python constructs also runs correctly without compilation. To do this, we add appropriate pure Python instance methods to the respective classes; if compilation fails, this pure Python implementation is run, ensuring that the user experience is not broken due to the compiler not being able to translate a construct, although the resulting computation may be orders of magnitude slower.

```
class My1DKernel(StencilKernel):
  def __init__(self):
    super(MyKernel, self).__init__()
    # set the neighbors set 1 as the point
    # before and the point after
    self.set_neighbors(1, [[1],[-1]])

  # the actual kernel function
  def kernel(self, out_grid, in_grid):
    for x in out_grid.interior_points():
      for y in self.neighbors(in_grid, x, 1):
        out_grid[x] += 0.5 * in_grid[y]

input_grid = StencilGrid([100])
output_grid = StencilGrid([100])

# load the data from somewhere
DataLoader().load(input_grid)

# apply the kernel
MyKernel().kernel(out_grid, in_grid)
```

Figure 10.3: Full example of defining a simple 1D kernel and calling it in Python, using Sepya.

10.2 Structure of the Sepya Compiler

The structure of the Sepya compiler follows the standard Asp flow described in Chapter 5. First, Python source is parsed and transformed into an abstract syntax tree (AST). Then, this AST is transformed into an instance of the Semantic Model, during which correctness checks are performed to ensure the computation can be correctly converted into low-level code.

After converting to the intermediate representation, the execution path diverges depending on whether the computation executes single or multiple timesteps. For a single-iteration kernel, the code is transformed into a set of nested loops in the C++ syntax tree, and these loops undergo phased transformations to apply a number of optimizations (see Section 10.3). These transformations primarily utilize built-in standard transformations from the Asp framework, as described in Chapter 5, but use domain-specific knowledge to control when and how to apply them, using insight gained from the proof-of-concept in Chapter 9. For example, the parallelization uses Asp's standard OpenMP support, but domain-specific knowledge dictates that the parallelization should not cut the least unit-stride dimension.

In the case of a multi-timestep computation, because the majority of the performance gain is due to blocking in time to reuse data already in cache, the algorithm is implemented by directly translating just the innermost computation (the function applied at each point). This transformed computation is then inserted into an Asp template, instantiated using the properties of the kernel, including the footprint of the computation and the dimensionality of the grids.

To facilitate debugging, the compiler also implements an interpreted backend, which can interpret each node in the intermediate representation instead of translating the computation. This allows implementing cross-language debugging algorithms, such as those that compare intermediate

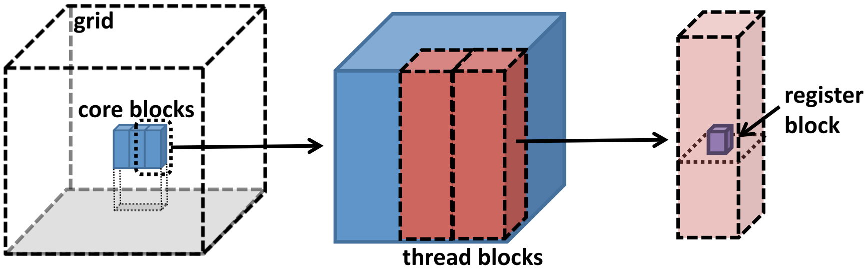


Figure 10.4: Parallelization/optimization strategy for the structured grid DSEL compiler. The grid is decomposed into core blocks, which are then decomposed into thread blocks, and finally into register blocks.

Optimization	Exploration in Original Tuner	Exploration in Sepya
NUMA-aware data allocation	√	√
Core blocking	2 most unit-stride dimensions	2^{nd} least unit-stride dimension only
Thread blocking	One per core block	One per core block
Chunking	√	-
Array index simplification	√	-
Array padding	√	√
Register blocking	All dimensions	Only most unit-stride dimension
Time skewing	-	√

Table 10.2: Summary of optimizations implemented in Sepya, compared to the proof-of-concept auto-tuner from Chapter 9.

states at synchronization points between the interpreted and translated versions, such as those described in Section 5.6. In addition, this interpreted backend also allows compiler developers to quickly test new front-end translations for new functionality without needing to implement full translation.

10.3 Implemented Code Generation Algorithms & Optimizations

Sepya implements two different algorithms, depending on whether the computation runs for a single iteration over the grid or for multiple iterations. This section describes the optimization strategies for the two algorithms. Table 10.2 summarizes the optimizations and parallelization used in the DSEL compiler and how they differ from the original proof-of-concept auto-tuner.

The single iteration algorithm, for structured grid computations that are not conducive to multi-timestep optimizations, is a variant of the multi-level blocking described in Chapter 9. For the DSEL compiler, we simplify the blocking by removing additional layers of blocking to make it more general, and to ease implementation difficulty. Although the removed layers of blocking improve performance in some cases, the trade-off is worth the reduced compiler complexity as long as the obtained performance is still a high percentage of peak, and applies to a larger class of structured grids.

The blocking is shown in Figure 10.4. A grid is decomposed into *core blocks*, each of which is further decomposed *thread blocks* and into *register blocks*. Each thread block is often small enough to fit into cache, so further cache blocking optimizations that target per-thread caches are generally not helpful, although at extreme sizes, this may change. At the lowest level, register blocks allow the computation to exploit registers (including SIMD) by explicitly exposing inter-iteration reuse and potential vectorization opportunities; this kind of blocking is implemented by unrolling the loops explicitly so the compiler can see the explicit reuse.

For parallelism, we use OpenMP to distribute the core blocks over the processors. As shown in the figure, the grids are never subdivided in the unit-stride dimension, because such blockings interfere with hardware prefetchers [61]. Furthermore, each thread block only reads from adjacent thread blocks, which allows us to exploit page placement policies on systems with non-uniform memory access (NUMA) to ensure as little communication as possible between memory controllers.

Currently, many of the optimizations are not applied when the dimensionality of the grid is less than three. In particular, 2D structured grid computations are only parallelized, not blocked, since the parallelization implicitly blocks the computation, and further blocking is usually not beneficial. In addition, 1D grids are not even parallelized, because this interferes with prefetcher performance and rarely speeds up computation.

Further optimizations could be implemented in the DSEL compiler. Although the incremental increase in performance could be low, further levels of blocking may be beneficial. Most of the current optimizations target memory-bound structured grid computations; additional optimizations targeting compute-bound kernels could be implemented, including explicit vectorization (often, compilers are unable to vectorize code due to possible dependencies). Such optimizations are a subject of possible future work.

For stencils with multiple timesteps, we use a parallel variant of the time-skewing stencil algorithm [122]. Our version subdivides the timespace of the computation into blocks, then works on individual blocks together; these blocks are further cache blocked using the same strategy as single-timestep computations. This avoids synchronizations that are required if each thread works on separate time-space blocks, but may limit the effectiveness of our parallelization. Currently, only 3D or higher-dimension kernels are optimized using time skewing.

10.3.1 Auto-tuning

Sepya leverages Asp's auto-tuning support. For the blocking strategies we implement for both single- and multi-timestep computations, the DSEL compiler creates a large number of parameterized variants, each with a different combination of optimizations. For a typical 3D test case, the number of variants is over 200.

Currently, the only search strategy implemented is the one from Asp: exhaustive search. Because the search is online, a different variant is run each time a kernel is called until all have been run, after which the fastest one is always used. More intelligent search strategies could reduce the search space.

10.3.2 Data Structure

To facilitate possible data structure transformations, we restrict the input and output data structures for our kernels to be instances of the provided `StencilGrid` class. Instances of this class provide

Kernel	Dimensionality	Connectivity	Boundaries	Unoptimized OI
Laplacian	3D	Rectahedral	Constant	0.20
divergence	3D	Rectahedral	Constant	0.14
gradient	3D	Rectahedral	Constant	0.11
hex-divergence	3D	Hexahedral	Constant	0.33
tri-smooth	2D	Triangular	Constant	0.09
bilateral-r1	3D	Rectahedral	Constant	$>1^\dagger$
bilateral-r3	3D	Rectahedral	Constant	$>1^\dagger$

Table 10.3: Test structured grid kernels used in this study. Recall that Operational Intensity (OI) is defined as $\frac{ops}{bytes}$. Those marked with † are approximate due to modulo and integer conversion operations.

pure Python implementations of the various iterators and functions used in structured grid kernels. This also allows the kernels to run even in the absence of translation and compilation, since we provide methods in our class that enable execution. In addition, we restrict the elemental datatypes to a subset of those supported by NumPy [88]: integers and double- and single-precision floating point numbers. These restrictions allow the compiler to generate typed C++ code from untyped Python, since the elemental types are known.

One major optimization for multi-socket machines is to exploit default page placement policies that dictate that pages will be pinned to memory controllers that first "touch" them on NUMA machines. When such a machine is detected, the default NumPy memory allocator is not called; instead, we generate C++ code that touches data carefully to ensure the page placement corresponds to the execution units that will perform the computation. This reduces unnecessary inter-socket communication due to misplaced memory pages.

We also implement padding of the data structures to ensure accesses occur at cache line boundaries. This decreases unnecessary cache traffic, and can also ensure that the code is more easily vectorizable by the optimizing backend C++ compiler.

10.4 Evaluation

In this section, we evaluate the obtained performance of the structured grid DSEL on a set of kernels obtained from actual applications. We then outline productivity gains due to expressing computations in a high-level manner, and show some possible improvements to the auto-tuning search.

10.4.1 Test kernels & Other DSL systems

We evaluate obtained performance by comparing our DSEL with Pochoir [105], an external stencil DSL that uses a parallel variant of the cache-oblivious stencil algorithm [40]. Written in Haskell, the Pochoir compiler is used to transform code in a C++ like DSL into a parallel version that uses Cilk Plus. The implemented algorithm can take advantage of multi-timestep computations by blocking in both time and space.

DSL	Type	Implementation Language	Output Language	Parallelism	Approx LoC
Graphite	Compiler Infrastructure	C	-	-	9300
Pochoir	External	Haskell	C	Cilk Plus	7280
Sepya	Embedded	Python	C++	OpenMP or Cilk Plus	1100

Table 10.4: Summary of structured grid systems compared in this chapter.

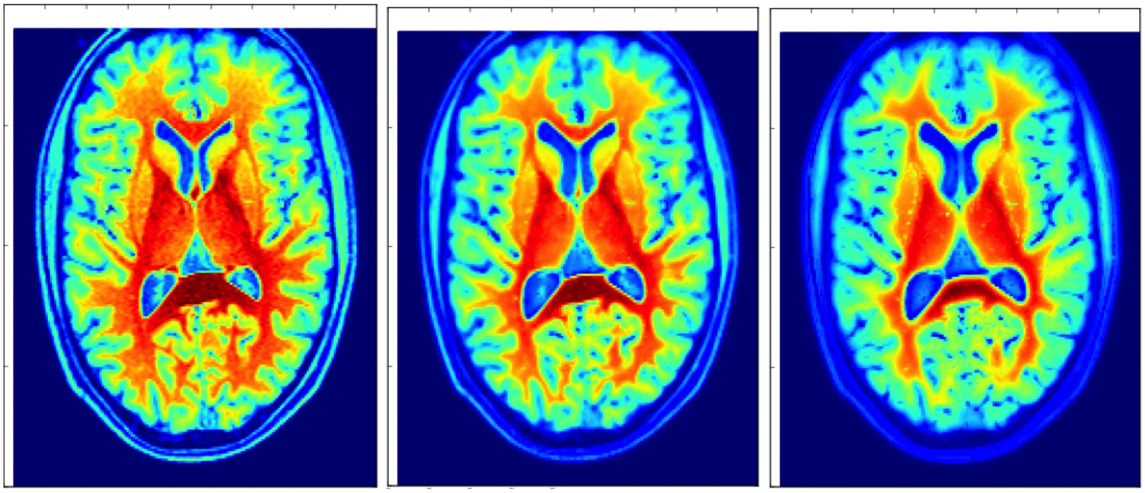


Figure 10.5: Example application of a bilateral filter. Left: a 2D slice of the input 3D dataset. Center: same 2D slice after applying a 3D bilateral filter with $r = 1$. Right: same 2D slice after applying a 3D bilateral filter with $r = 3$.

The two DSL systems are compared with results obtained by using Graphite [94], which provides compiler support for optimizing stencil-like loops in the Gnu Compiler Collection using the polyhedral model. The polyhedral model transforms nested loops into polyhedra with dependence information, and uses this abstraction to determine an efficient and correct traversal, including parallelization. A summary of features of the three systems is shown in Table 10.4.

The set of test kernels is summarized in Table 10.3, including dimensionality and operational intensity. The kernels are run with sizes of 4096^2 and 256^3 for 2D and 3D respectively, not including ghost zones. Some salient features of the kernels are described below.

Laplacian, Divergence, Gradient. These are three standard 3D kernels from the literature, which are normally bound by memory bandwidth according to the Roofline model. Because they vary in the number of input and output grids, each has different operational intensity. The Laplacian kernel is suitable for running for multiple timesteps.

Bilateral Filter. The 3D bilateral filter kernel comes from a medical imaging application and is used to reduce noise and enhance important features in magnetic resonance imaging (MRI) of the brain [77]. Unlike a Gaussian filter, the bilateral filter combines spatial and signal weights to create an edge-preserving smoothing filter. Our application requires applying the filter with varying radii to highlight features of different sizes; an example of applying the filter to actual MRI images is shown in Figure 10.5. We use $r = 1$ and $r = 3$ filter radii, which have quite different operational

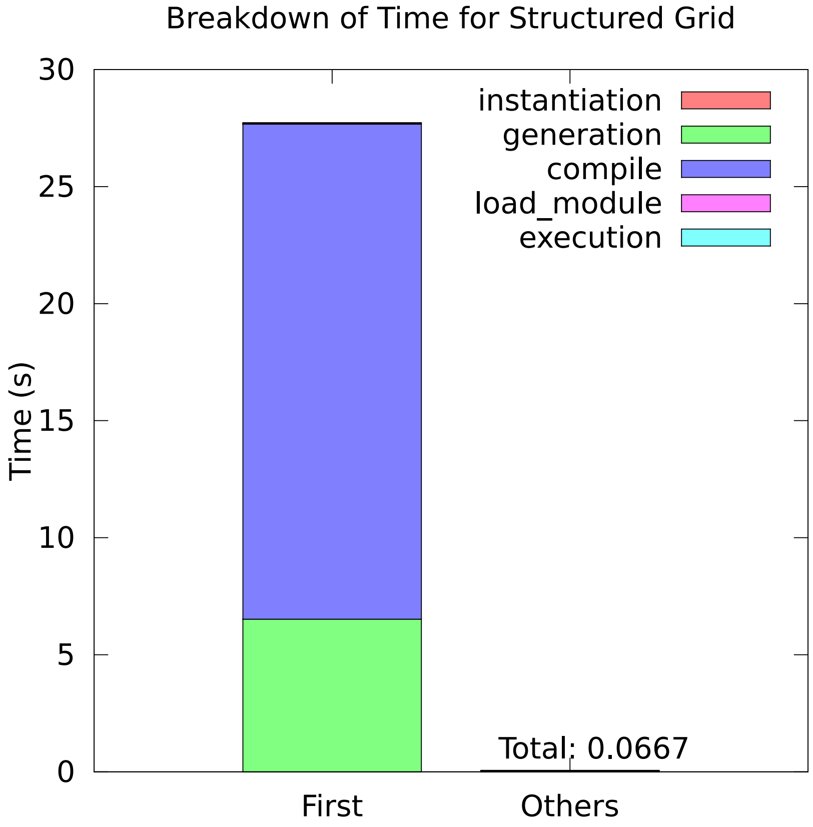


Figure 10.6: Breakdown of overheads for the structured grid DSEL, using the 3D Laplacian kernel as an example. On first load, over 6 seconds are spent on generating the large number of optimized variants, and over 20 seconds are spent invoking the compiler. For subsequent runs, instantiating the class and loading the already-compiled module takes a negligible amount of time due to caching the compiled code.

intensity; at $r = 1$, the filter is a 27-point stencil, while at $r = 3$ it becomes a 343-point stencil. In both cases, there is so much computation that the kernels are bound by computation rate.

Hexagonal Divergence, Triangular Smoother. These are both 2D kernels, one of which comes from a next-generation climate simulation code, and the other from a proprietary application. The primary purpose of selecting these kernels is to see how the various structured grid systems fare for non-rectahedral topologies. Like most of the test kernels, these are also bound by memory bandwidth performance.

10.4.2 Breakdown of Execution Time

Compared to a statically-compiled language, the dynamic nature of the SEJITS approach incurs some overheads due to code generation, compilation, and loading a compiled dynamic link library into the Python interpreter. Figure 10.6 shows the relative costs of each overhead for both the initial run, and for subsequent invocations of the interpreter. Note that there is zero overhead once the library has been loaded.

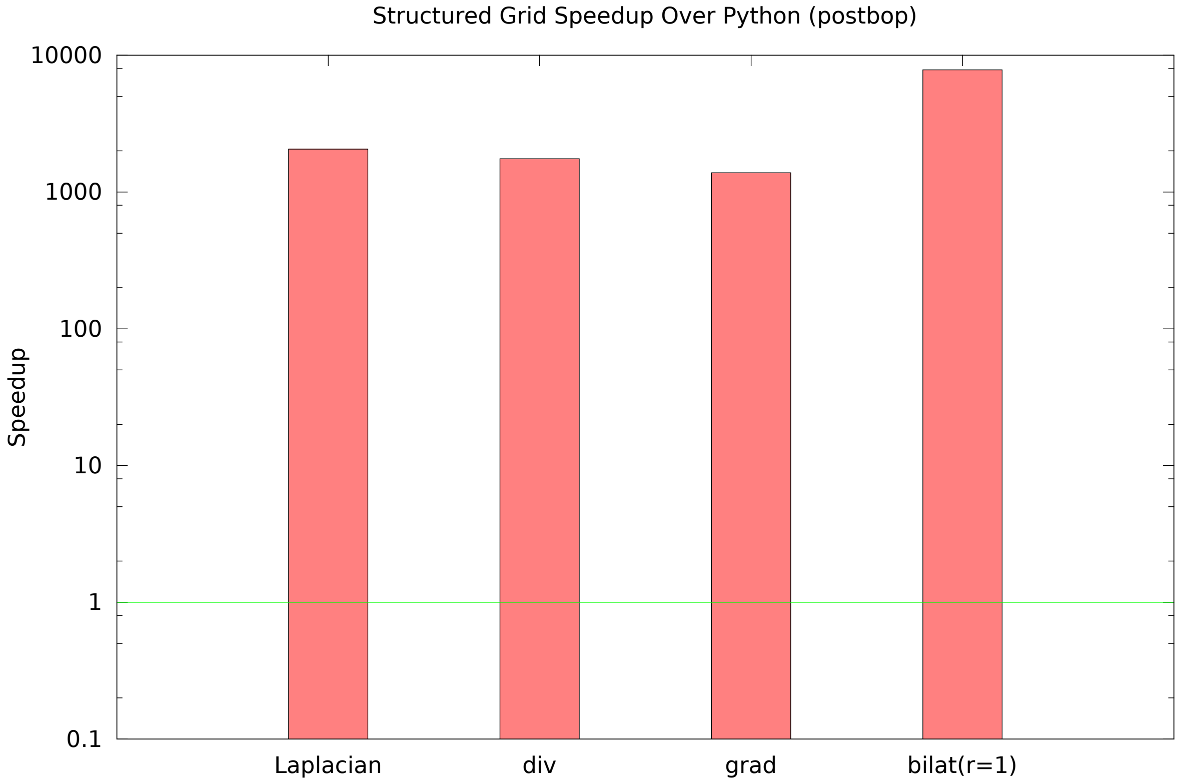


Figure 10.7: Performance for selected kernels versus pure Python on Postbop.

By far, the largest cost, 21 seconds, is due to running an external compiler, which is not optimized for speed. Using minimal compilers such as the Tiny C Compiler (TCC) [12] is one way to mitigate this cost, but would come at a large cost to performance, since much of the time taken by the compiler is due to optimization passes. Generating the large number of variants (in this case, 227 variants) takes about 6 seconds. Loading in the library, which is done using Python's standard `import`, takes over 36ms, and the execution in this case is 27ms.

For the rest of this chapter, these overheads are not reported, since they occur only the first time the computation is transformed and compiled, or occur at module load time, which is not part of the execution. Furthermore, we do not report auto-tuning time; the idea is that after many executions (on the order of 200 for most of these kernels), the kernel will always use the fastest version, so we only report that version's performance as it represents the steady state.

10.4.3 Single Iteration Performance

Performance relative to NumPy implementations of the structured grid kernels is shown in Figure 10.7 on Postbop, the single-socket Intel Core i7 test machine. Sepya obtains performance that is 2–3 orders of magnitude faster than pure Python for every kernel, demonstrating the high performance possible with the SEJITS approach. For the rest of this section, we compare obtained performance with that of Pochoir and Graphite.

A summary of performance relative to Graphite is shown in Figure 10.8 on Postbop for both our

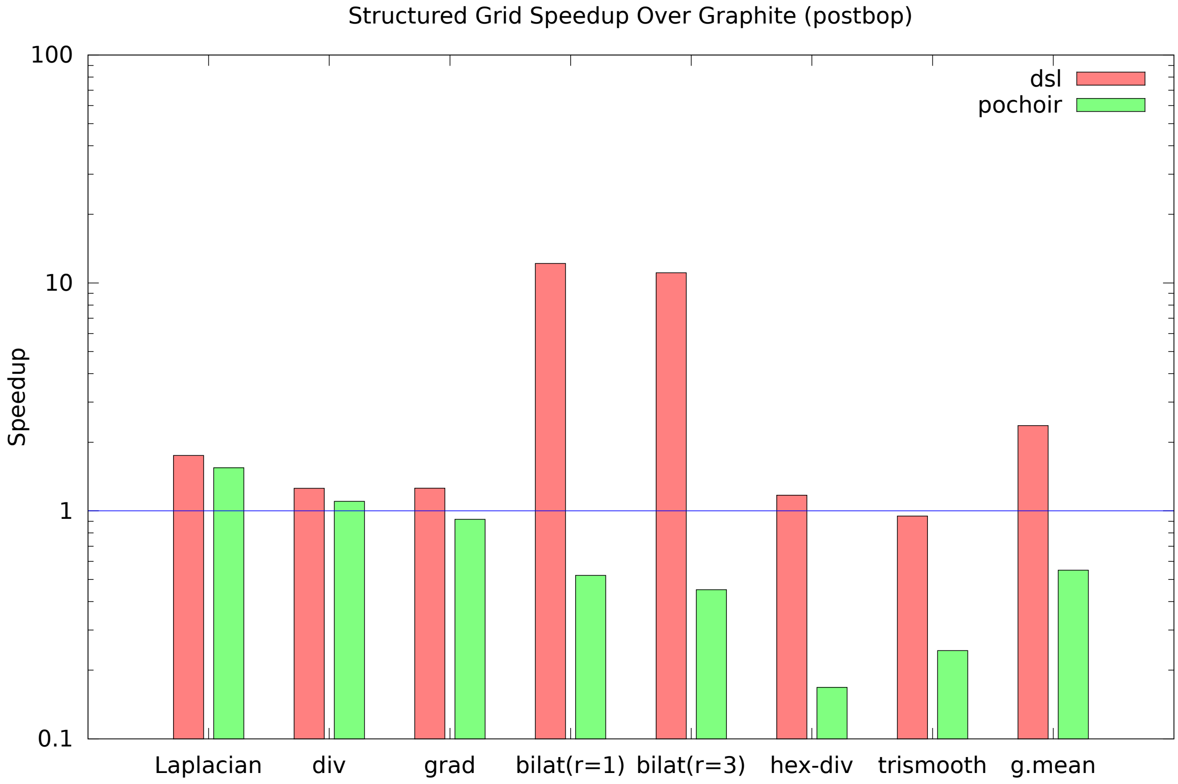


Figure 10.8: Performance for single-iteration kernels on Postbop.

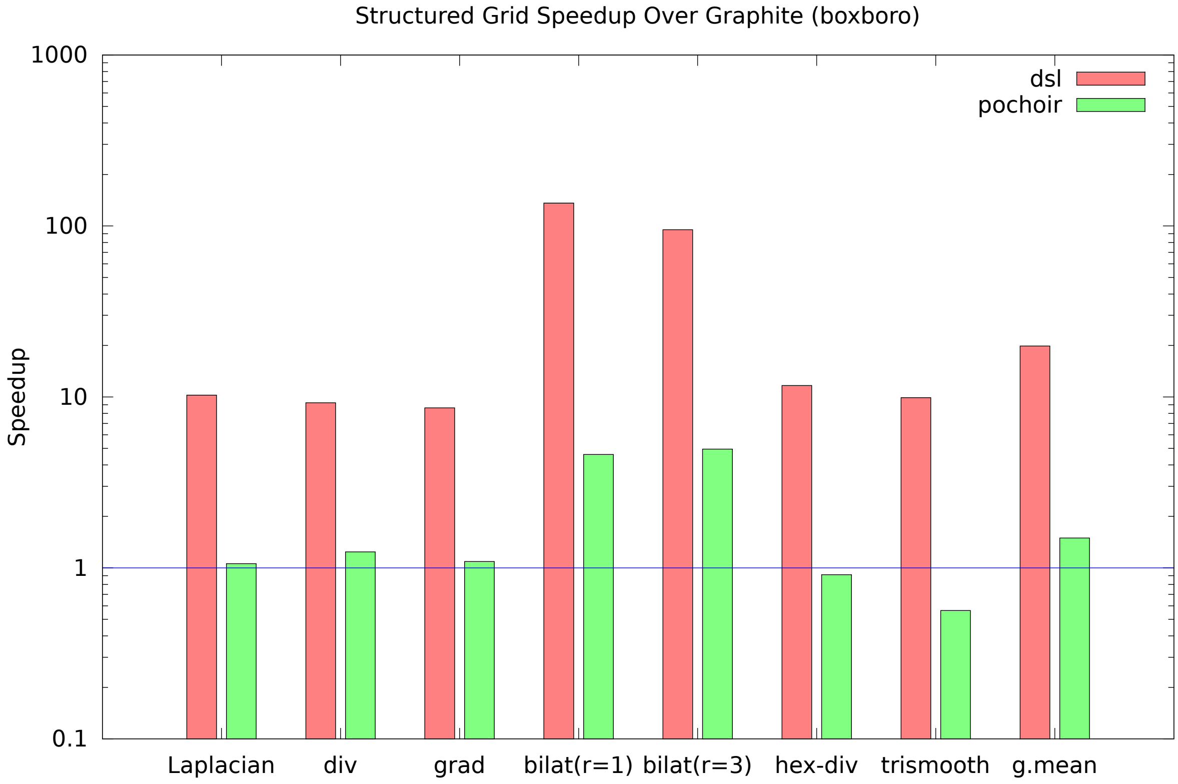


Figure 10.9: Performance for single-iteration kernels on Boxboro.

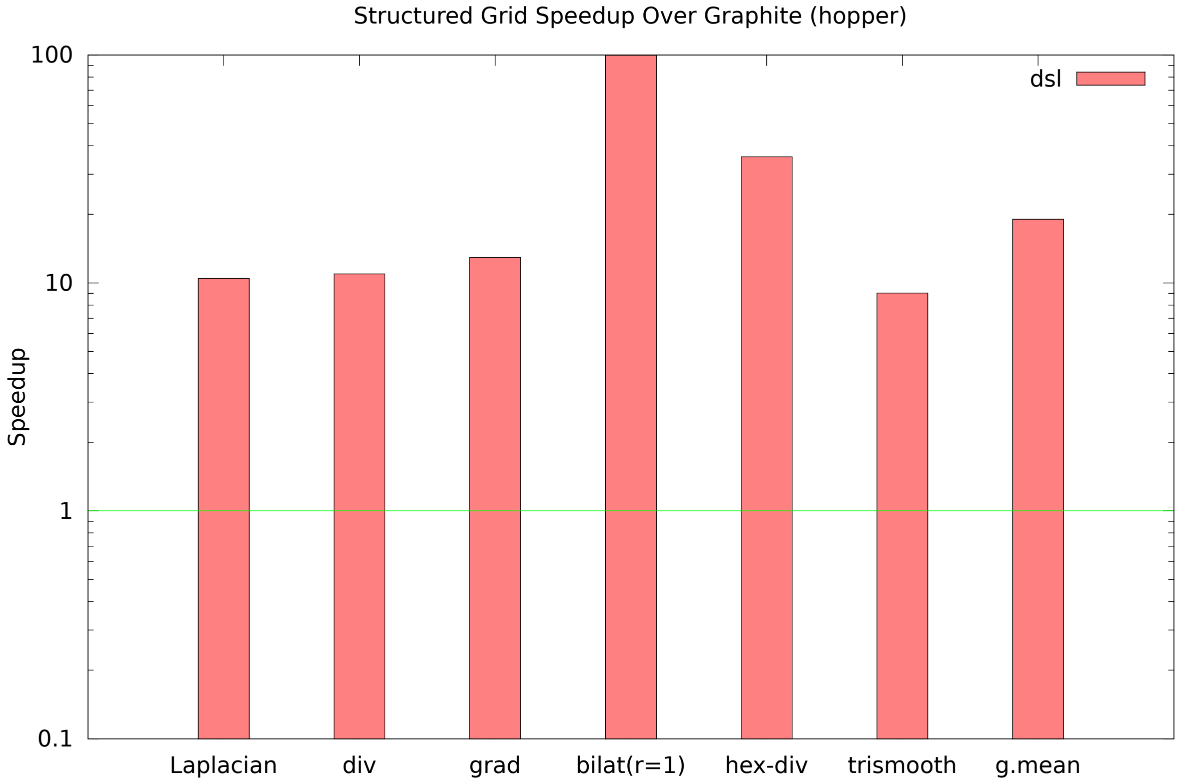


Figure 10.10: Performance for single-iteration kernels on Hopper.

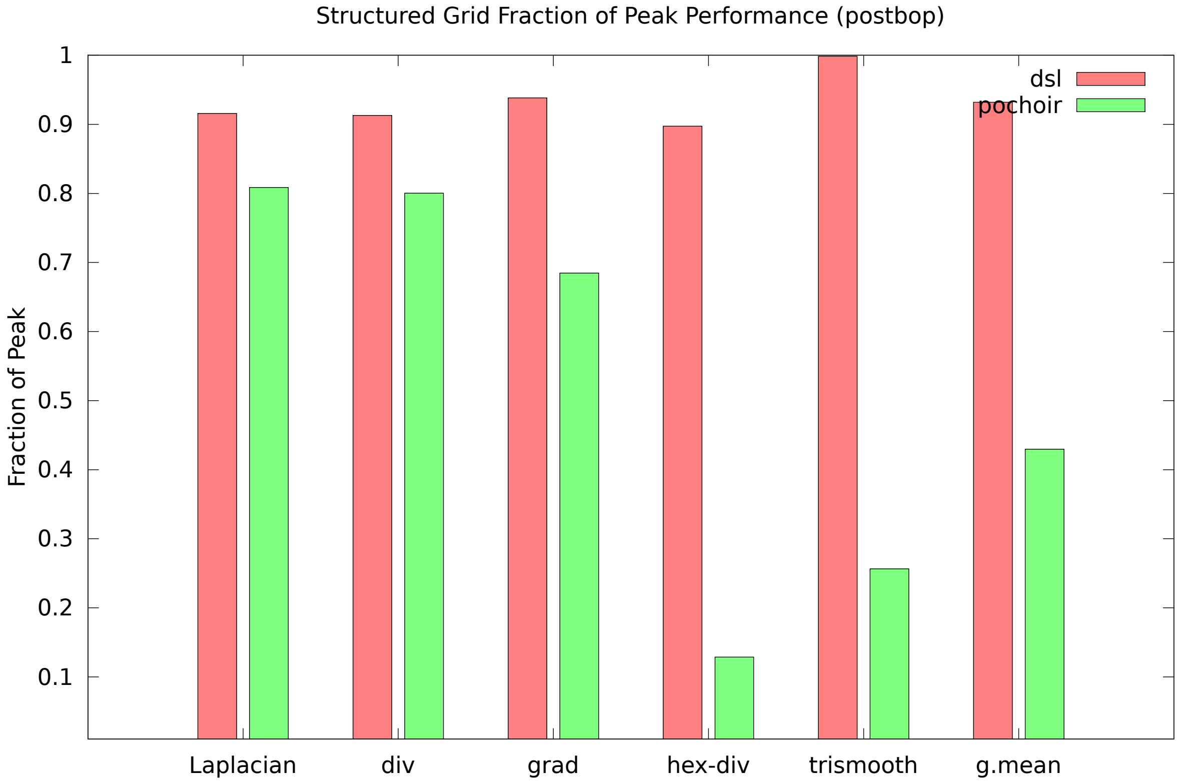


Figure 10.11: Performance as fraction of Roofline peak for single-iteration kernels on Postbop.

stencil DSEL and Pochoir. For all but one kernel, Sepya outperforms the polyhedral framework in Graphite. Relative to Pochoir, Sepya is able to obtain higher performance in every case due to our use of auto-tuning and cache-aware single-iteration optimizations. In contrast, Pochoir concentrates more on multi-timestep optimizations. In addition, Pochoir is hampered by its very poor performance for the bilateral filter, regardless of radius. The geometric mean of speedup relative to Graphite is about $2.37\times$ for our DSEL, while Pochoir's performance is about $0.55\times$ that of Graphite on this test machine.

On Boxboro, with its 40 cores and 10 sockets, Sepya performs even better relative to Graphite. Figure 10.9 shows the performance relative to GCC's polyhedral framework. In every case, our DSEL outperforms both Graphite and Pochoir, leading to a geometric mean of speedup of $19.9\times$ compared to Pochoir's $1.52\times$ speedup. NUMA-aware allocation is the most important optimization on this platform, and since neither Pochoir nor Graphite implement it, their performance suffers greatly.

For the AMD-based Hopper machine, we tested the available compilers and use the installed Intel compiler as it obtains the best performance for our studied structured grid kernels. The performance for our test kernels on Hopper is shown in Figure 10.10. Because there is no working Cilk Plus runtime on the system, we do not report Pochoir results, and, in addition, the Intel compiler was unable to compile the bilateral filter with radius 3 due to an internal error. Overall, the DSEL obtains a geometric mean speedup of $19\times$ over Graphite.

To compare the obtained performance with the best possible performance, we use the Roofline

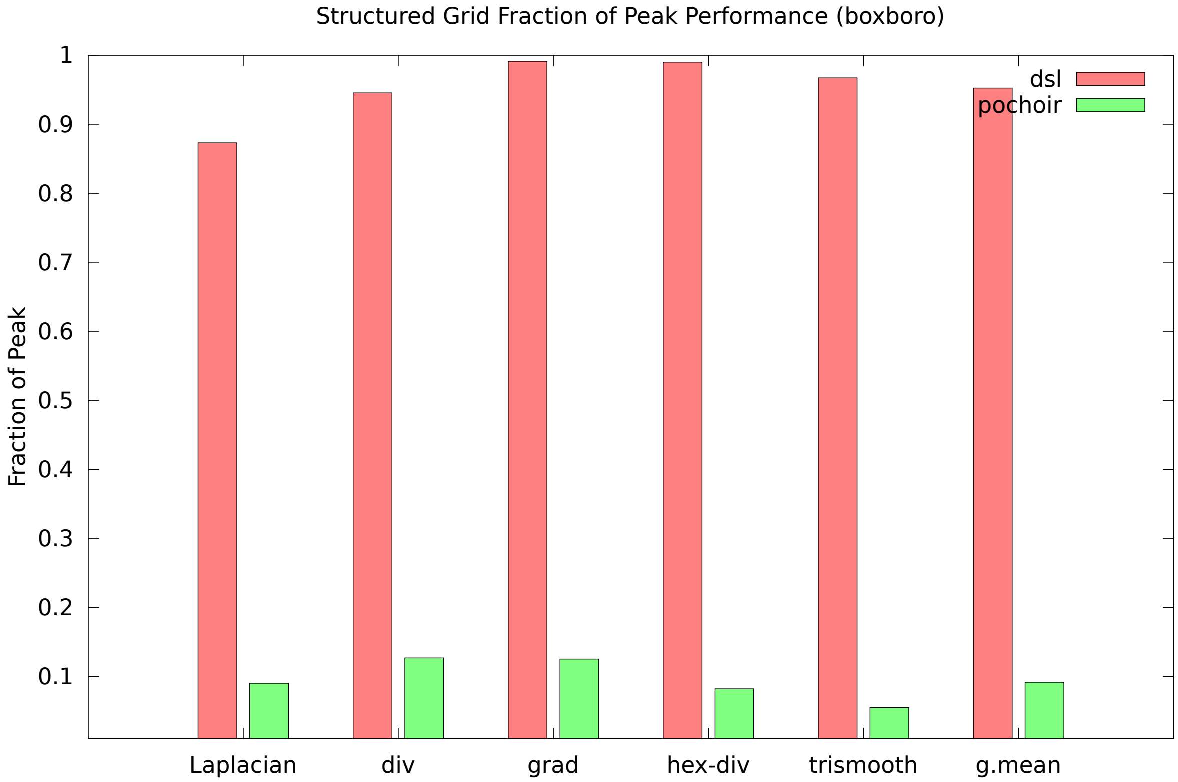


Figure 10.12: Performance as fraction of Roofline peak for single-iteration kernels on Boxboro.

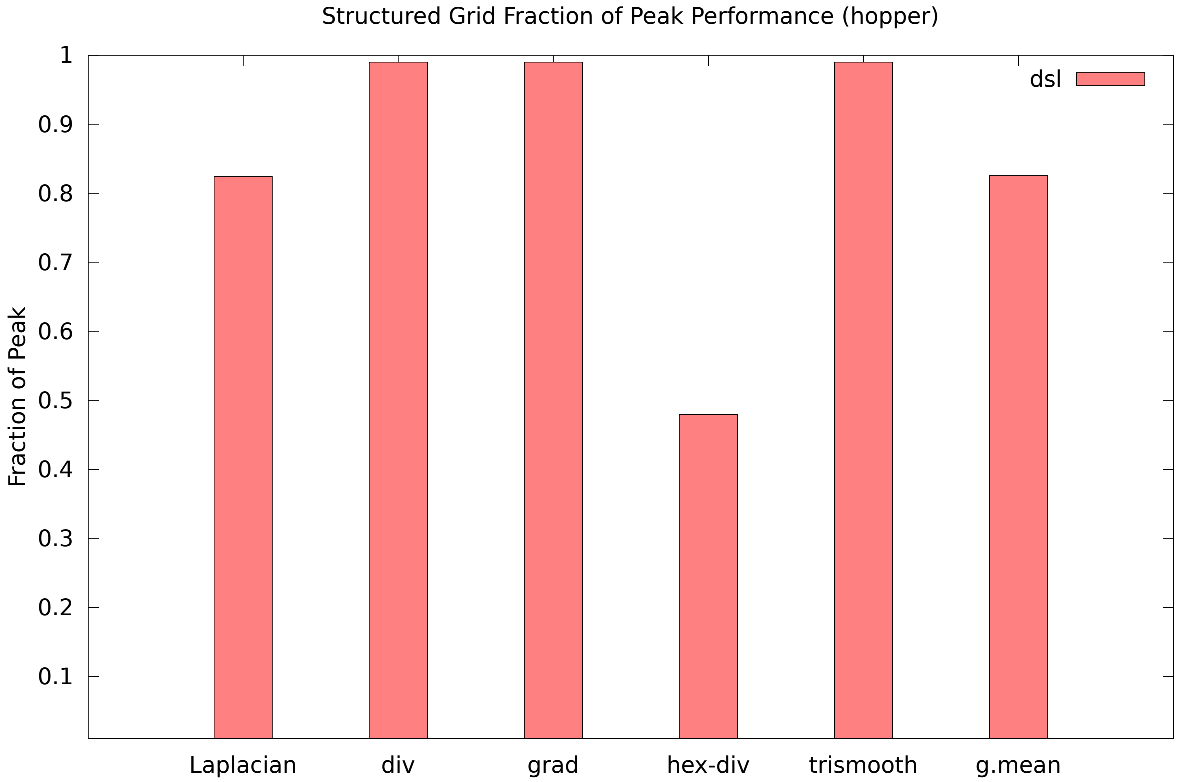


Figure 10.13: Performance as fraction of Roofline peak for single-iteration kernels on Hopper.

Kernel	Read Streams	Write Streams
Laplacian	2	1
divergence	4	1
gradient	4	3
hex-divergence	5	1
tri-smooth	2	1

Table 10.5: Read and write streams for memory-bandwidth bound kernels, including read streams for the output grids. The kernels are implemented using a read of the output grid as well.

model to determine the expected attainable performance. For most of these kernels (with the exception of the bilateral filters), the Roofline model shows that they are bound by off-chip memory bandwidth. Furthermore, on these architectures, memory bandwidth varies by the number of read and write streams in the computation; for example, a single write stream to memory will obtain lower bandwidth than multiple read streams.

We use a customized version of the STREAM [76] memory bandwidth benchmark to determine obtainable bandwidth. Our customized version matches the number of streams for each structured grid kernel, and executes using OpenMP parallelization. Table 10.5 summarizes the memory streams of each structured grid kernel and the proxy customized STREAM benchmark used for calculating attainable memory bandwidth. Arrays are sized larger than the largest cache, and initialized using NUMA-aware initialization. Note that in calculating the memory bandwidth obtained for the different stencils, we must include the ghost zones in both the read and write grids, because they are actually read and written to, due to the cache line granularity of reads and writes from the memory system.

Performance as a fraction of Roofline peak for the memory bandwidth bound kernels on Postbop is shown in Figure 10.11. On this single-socket architecture, the kernels perform at very high fractions of peak obtainable performance, with a geometric mean of 93% of peak, compared to Pochoir's 42% of peak. This is an incredibly high fraction of peak, and points to little value in further optimizing the code generator for single-socket machines.

On Boxboro, as shown in Figure 10.12, the geometric mean is 95% of peak, compared to Pochoir's 9% of peak. The performance once again is exceptional, and displays the benefits of controlling both the data structure and the computation; this allows us to match the NUMA-aware allocation of the grids with their NUMA-aware computation.

Figure 10.13 shows the performance of the memory-bound kernels as a fraction of memory-bandwidth peak on Hopper. Note that for many of the kernels, the memory subsystem is operating at 98–99% of obtainable peak. However, for the hexahedral divergence kernel, performance is especially poor due to poor SIMDization. Nevertheless, the memory bound kernels obtain a geometric mean of over 83% of peak.

10.4.4 Multiple Iteration Performance

Sepya implements both single and multiple timestep algorithms, blocking in both space and time for the latter. Pochoir also implements space-time blocking for multiple timestep algorithms using its cache oblivious approach. Figure 10.14 shows the performance of the 3D Laplacian on a 258^3 grid

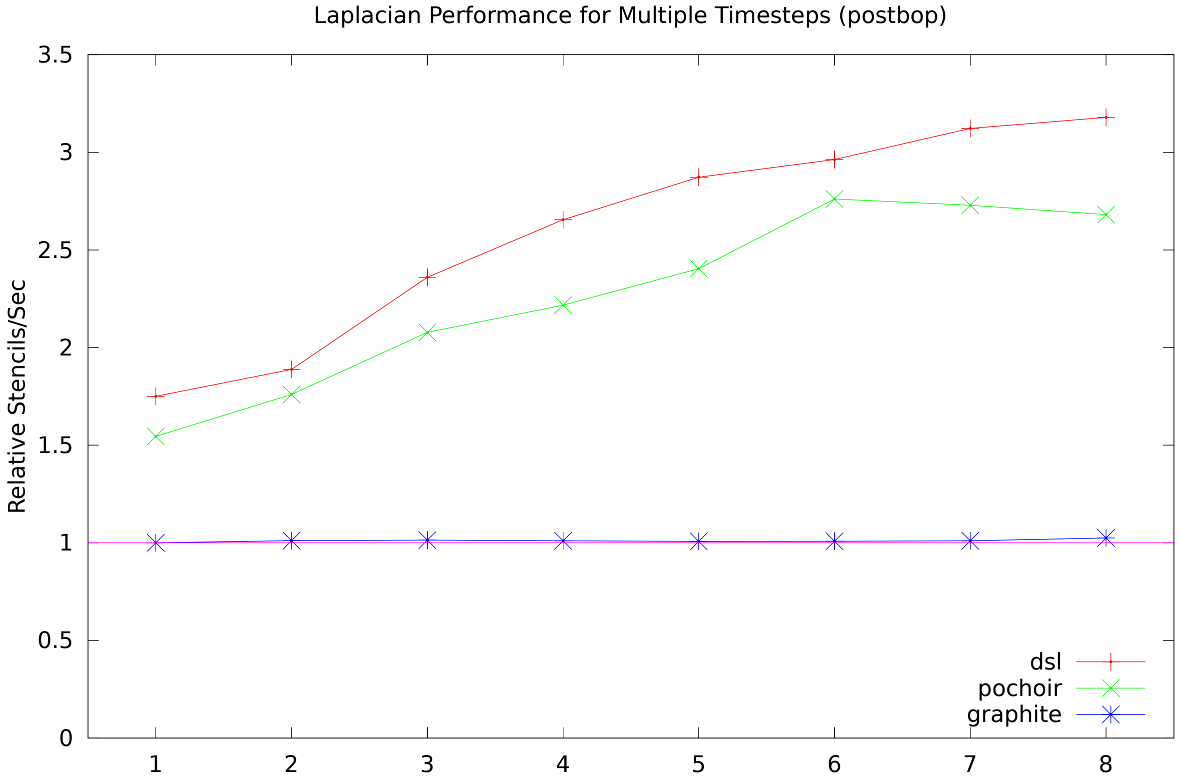


Figure 10.14: Multiple iteration performance on Postbop for the 3D Laplacian kernel on a 258^3 grid. Performance is normalized to one iteration of Graphite.

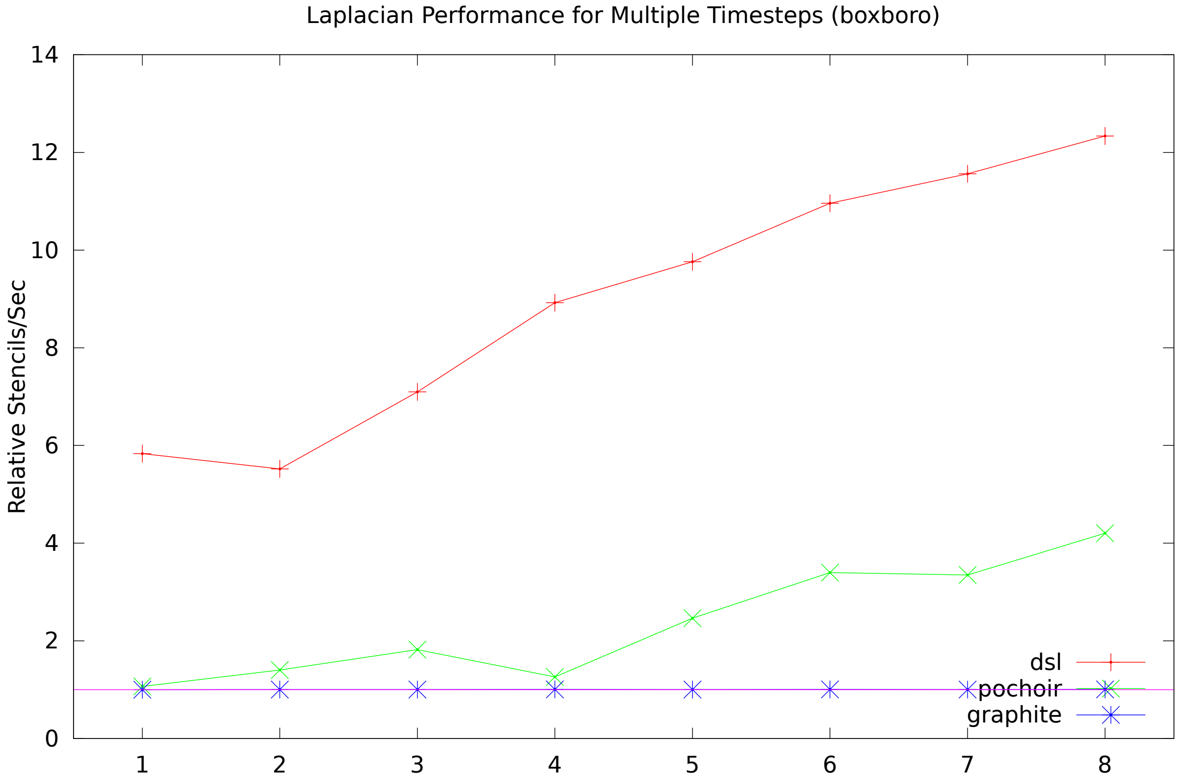


Figure 10.15: Multiple iteration performance on Boxboro for the 3D Laplacian kernel on a 258^3 grid. Performance is normalized to one iteration of Graphite.

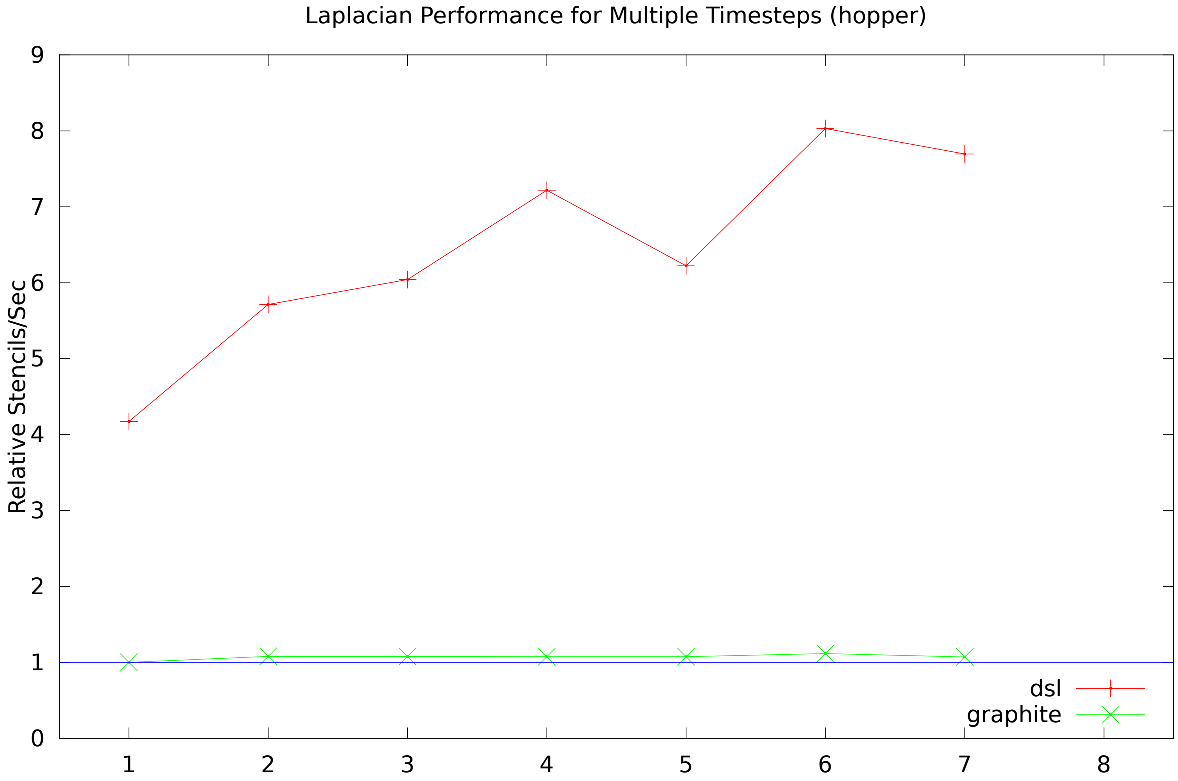


Figure 10.16: Multiple iteration performance on Hopper for the 3D Laplacian kernel on a 258^3 grid. Performance is normalized to one iteration of Graphite.

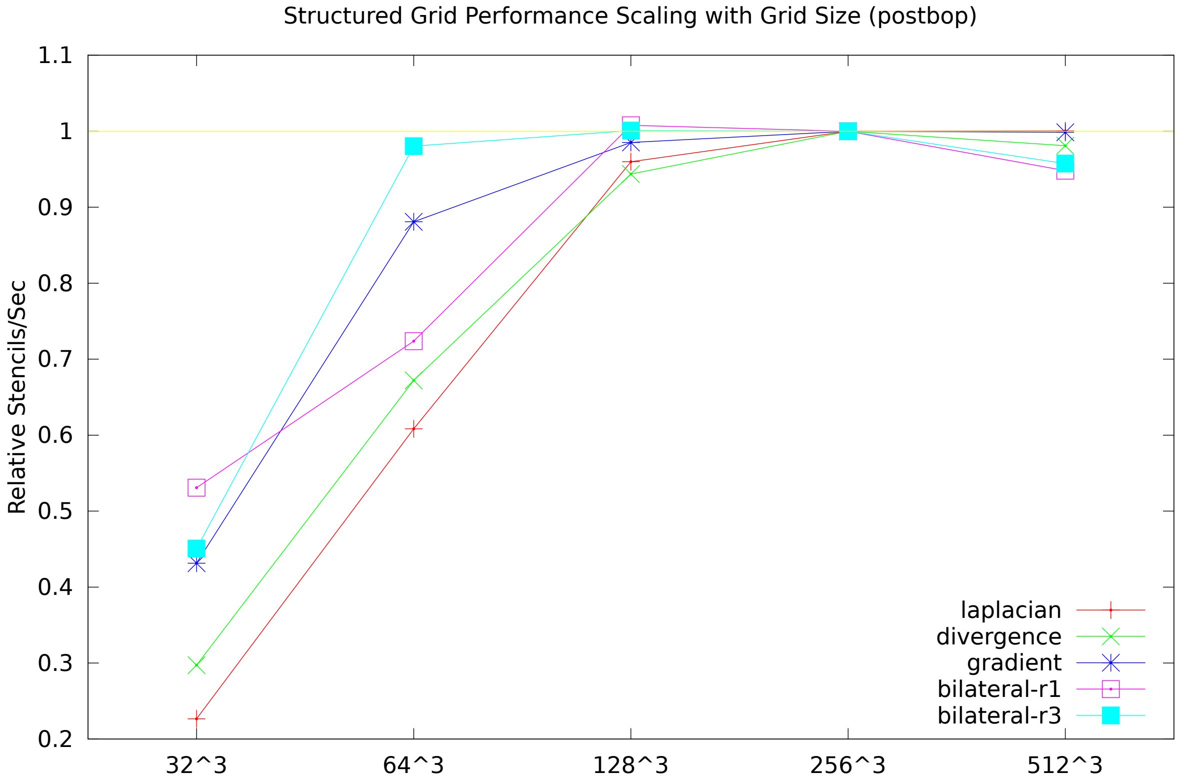


Figure 10.17: Performance as grid size is varied for the 3D kernels on Postbop. Here, performance is normalized to that obtained at 256^3 (not counting ghost zones).

on Postbop, normalized to a single iteration of Graphite. On this machine, the relative performance of the DSEL continues to increase, due to the space-time blocking, which reduces the overall cache misses by keeping data in cache while performing multiple timesteps of the calculation. Pochoir's relative performance, though greater than Graphite, levels off after 7 timesteps.

Performance for multiple timesteps on Boxboro is shown in Figure 10.15. The performance difference seen for a single timestep actually expands as we increase the number of timesteps, as Pochoir and Graphite both suffer from NUMA effects. In this case, both Pochoir and our DSEL show continuing performance increases as the number of timesteps increases, but Pochoir remains low relative to our DSEL.

On Hopper, as shown in Figure 10.16, we see results similar to Boxboro. The relative stencils per second is still increasing at the largest number of timesteps, and the gap between Graphite's non-time-blocked algorithm and our time skewing implementation is increasing. Pochoir results are not available due to the lack of a working Cilk Plus runtime on Hopper.

10.4.5 Grid Size Scaling

Our cache blocking scheme used in the DSEL compiler attempts to eliminate capacity misses caused by grids that are too large to fit in cache. One way to characterize its effectiveness is to look at how performance changes as grid size increases. Figure 10.17 shows the performance as grid size is increased, normalized to our 256^3 (not including ghost zones) test case. As expected, performance increases until it reaches a limit at about 128^3 and further increases in grid size affect little change on performance, thanks to blocking. Similar results occur on our other two test machines.

10.4.6 Expressibility

It is important to compare what kinds of structured grid computations can be expressed in the systems under study. Of the three systems, Graphite is the most flexible in terms of what kinds of computations can be expressed. Graphite can optimize almost any (perfect and imperfect) nested set of loops using the polyhedral method, as long as it can determine the dependence information.

Sepya and Pochoir both only handle Jacobi-style structured grid kernels, with both requiring the user to explicitly specify the stencil shape. In contrast, Graphite infers the dependence information from the program text. Pochoir and Sepya both require integer offsets in the stencil, disallowing variables in the stencil shape. Both also require the input and output grids be the same dimensionality and size (although both allow for lookups into grids with other dimensionality for weights); this precludes the two systems from expressing multigrid prolongation or restriction or other kinds of commonly-used kernels that have different sizes for the inputs and outputs.

Pochoir allows expressing boundary calculations and optimizes them by inlining them into the grid traversal. In contrast, Sepya currently does not optimize boundary calculations (except constant boundaries) and runs them slowly in serial pure Python.

Overall, the two DSEL systems express and execute similar types of structured grid problems, despite their different parallelization strategies and the fact that Sepya uses auto-tuning and is embedded in a high-level language.

10.4.7 Programmer Productivity

Productivity is difficult to measure without user studies, but lines of code are often used as a rough proxy. Comparing Graphite to our structured grid DSEL, we find that most kernels are about the same length in both, except for the bilateral filter kernels, which we can express more concisely by using Python's powerful standard library to define the neighbors. However, when compared to Pochoir, defining the structured grid kernel is much shorter in our DSEL, both thanks to our domain-specific constructs and thanks to the conciseness of Python relative to C++ .

The productivity of the performance programmer, who designs and implements the optimization frameworks, is also important. As seen in Table 10.4, our DSEL is one seventh the size of Pochoir and even smaller relative to Graphite. Much of this is due to our use of the Asp infrastructure, which handles auto-tuning as well as defining many commonly-used transformations. Thus, our DSEL demonstrates that this framework-based approach leads to productivity for both the performance programmer and the DSEL user.

10.4.8 Improving Auto-tuning Search

The auto-tuning search provides up to 4× increase in performance. However, the large number of candidate implementations means the search takes many runs to finish before the auto-tuner decides on the best implementation. In order to limit this combinatorial explosion, we first eliminate parameter sets that are unlikely to yield good performance; for example, we eliminate block sizes that are too small in the unit-stride dimension as previous work has shown such blockings are never close to optimal, due to their poor prefetching performance [60].

Asp should support more intelligent searches using machine learning. However, even in the absence of such infrastructure support, we can explore whether even slightly more intelligent searches would result in quickly converging to the best without trying all the possible variants. Previous work in other domains [116] has shown that auto-tuning search can be a “needle-in-haystack” type of problem; that is, the best variant may be surrounded by poorly-performing variants. However, the serial performance model in [60] gives hope that obtaining local maxima of performance is possible using a simple hill climbing or gradient descent algorithm; these local maxima can be close to optimal.

We use a variant of hill climbing called *random-restart hill climbing*. The search proceeds by selecting a random start location in the tuning space; subsequent runs search the neighborhood of this location for a variant that performs better. If one is found, the “vector” in the tuning space pointing from the original to the faster variant is followed stepwise with each subsequent call of the stencil. If no faster variant is found in the neighborhood of the central location, a new starting point is chosen at random. The search continues until either a set number of variants have been explored or no more variants are available.

Figures 10.18–10.20 shows the results of the hill climbing search on Postbop. For most kernels, we can obtain 98% of the best variant’s performance by searching 25% of space in the worst case; much of this is due to our pre-pruning of variants that will definitely not yield high performance.

Figures 10.21–10.22 show the results of applying the random-restart hill climbing algorithm to find best parameter values on Boxboro. Due to the NUMA nature of the machine, incorrect parameter selections can have a very large impact on performance. In the worst case, up to 50% of the space must be searched to get within 90% of the best obtained performance, but this is still better than exhaustively searching the entire space.

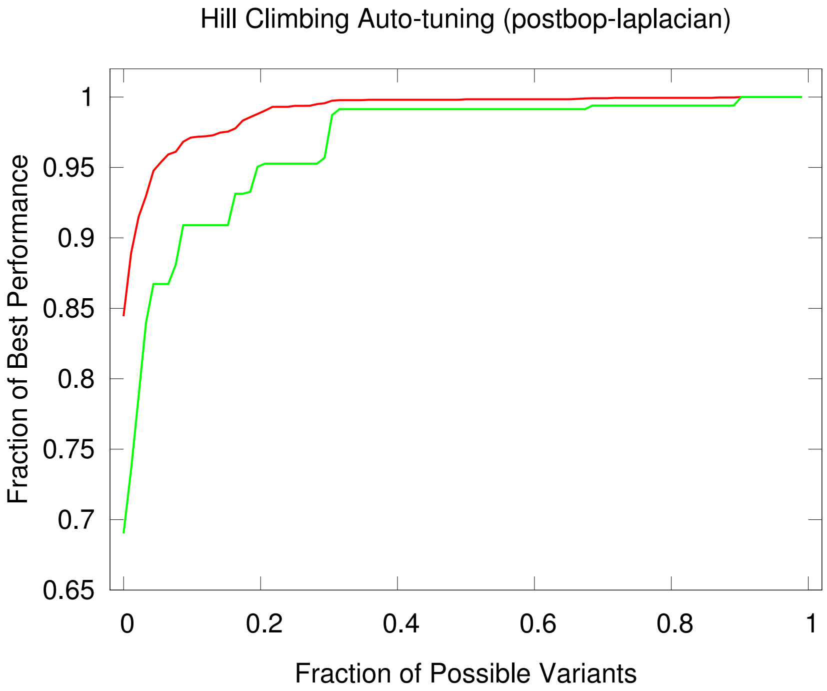
10.5 Future Work

The previous section describes a potential way to improve our auto-tuning strategy by using more intelligent search. In this section, we outline some further improvements that can be made to our DSEL and compiler to increase performance as well as expressibility.

10.5.1 Language Extensions

The current restriction that input and output grids have the same topology prevents expressing some important structured grid algorithms, namely kernels present in multigrid and Adaptive Mesh Refinement (AMR). In these algorithms, information is conveyed from one grid to another, and grids may have different connectivities. For example, in the prolongation kernel in multigrid, the output grid has more points than the input grid (usually a $2\times$ as many in each dimension).

Figure 10.24 shows example prolongation and restriction operators, using grid stepping constructs inspired by the Titanium [124] parallel programming language, in which the interior iterators can have a stride. For example, accessing a grid using a stride of 2 would yield every other point. At the same time, the neighbor iterator still accesses neighbors based on the indexing of the grid passed to it. A second extension allows iterating over neighbor points of the output grids. Because



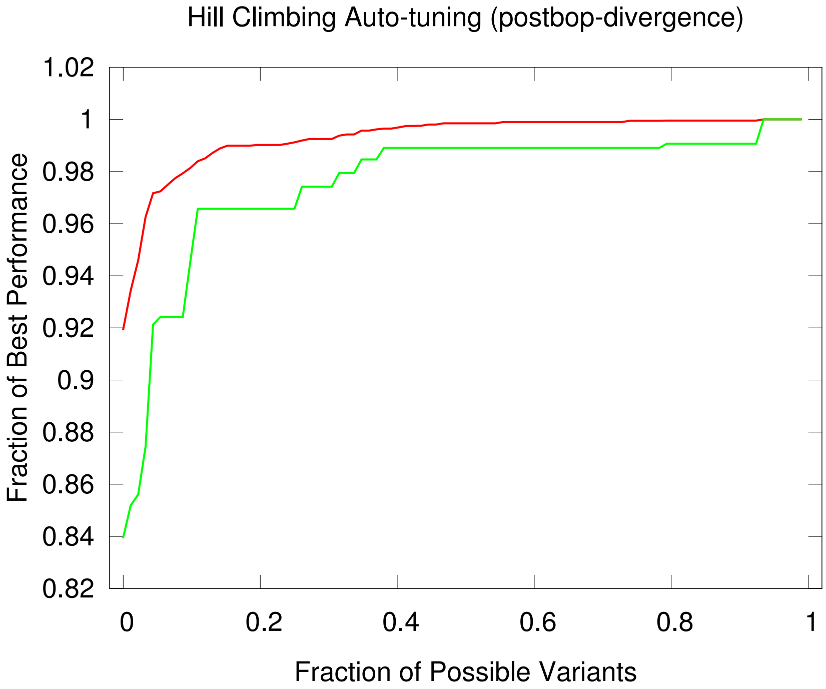
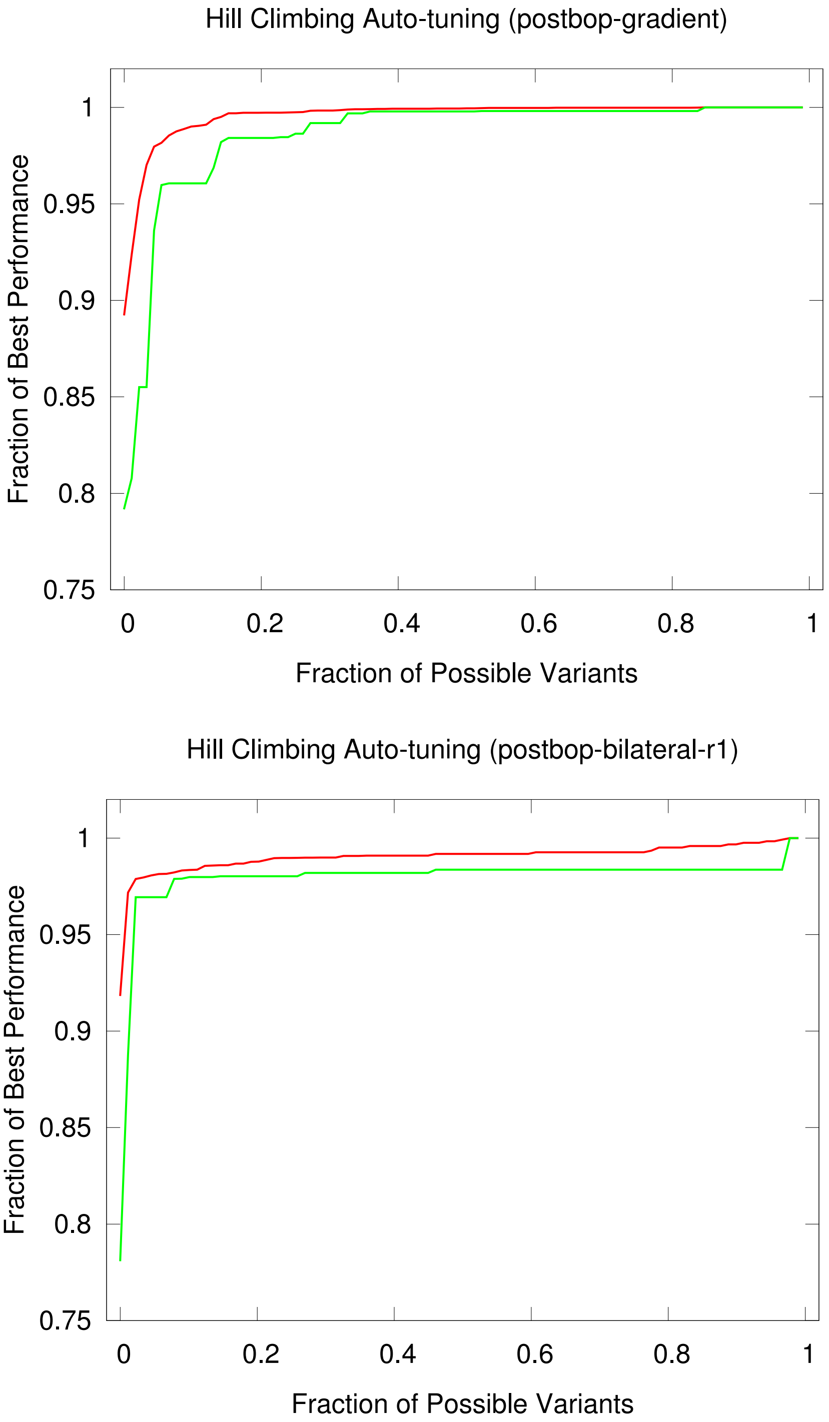


Figure 10.18: Hill climbing experiment on Postbop, for 3D kernels running on a 256^3 grid for a single iteration. The top line shows the average of 25 runs of the experiment, while the bottom line shows the worst run.


Hill Climbing Auto-tuning (postbop-gradient)

Hill Climbing Auto-tuning (postbop-bilateral-r1)

Figure 10.19: Hill climbing experiment on Postbop (continued), for 3D kernels running on a 256^3 grid for a single iteration. The top line shows the average of 25 runs of the experiment, while the bottom line shows the worst run.

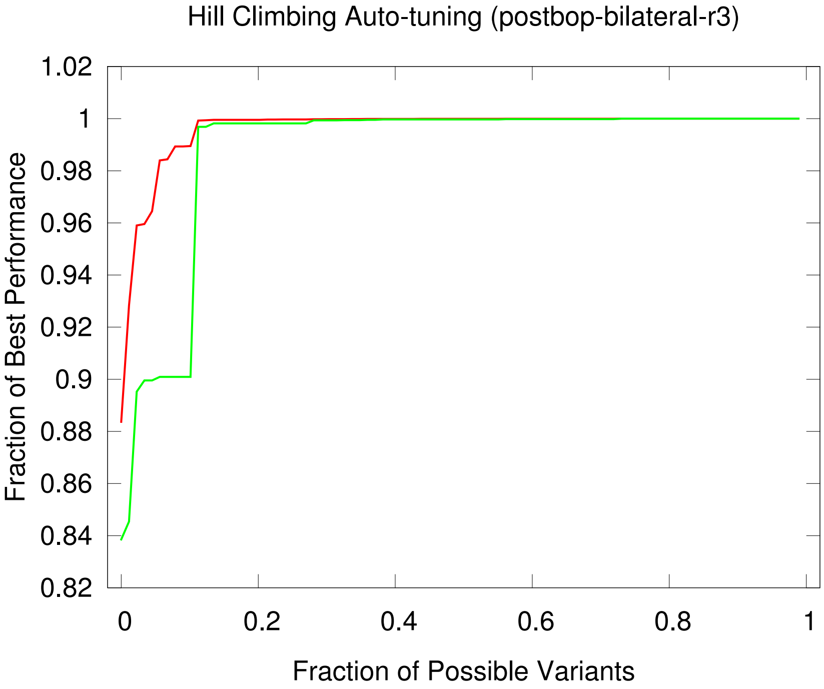
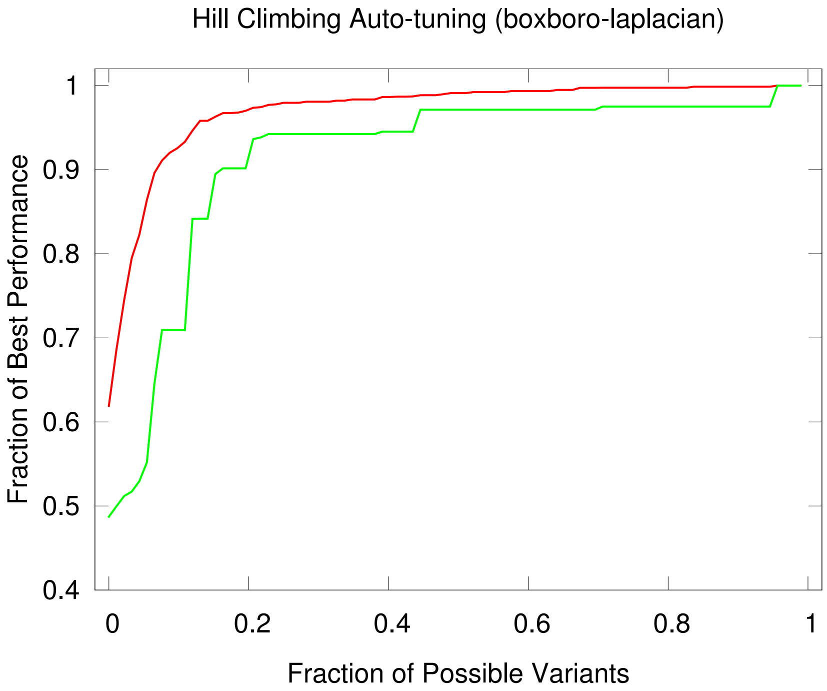


Figure 10.20: Hill climbing experiment on Postbop (continued), for 3D kernels running on a 256^3 grid for a single iteration. The top line shows the average of 25 runs of the experiment, while the bottom line shows the worst run.



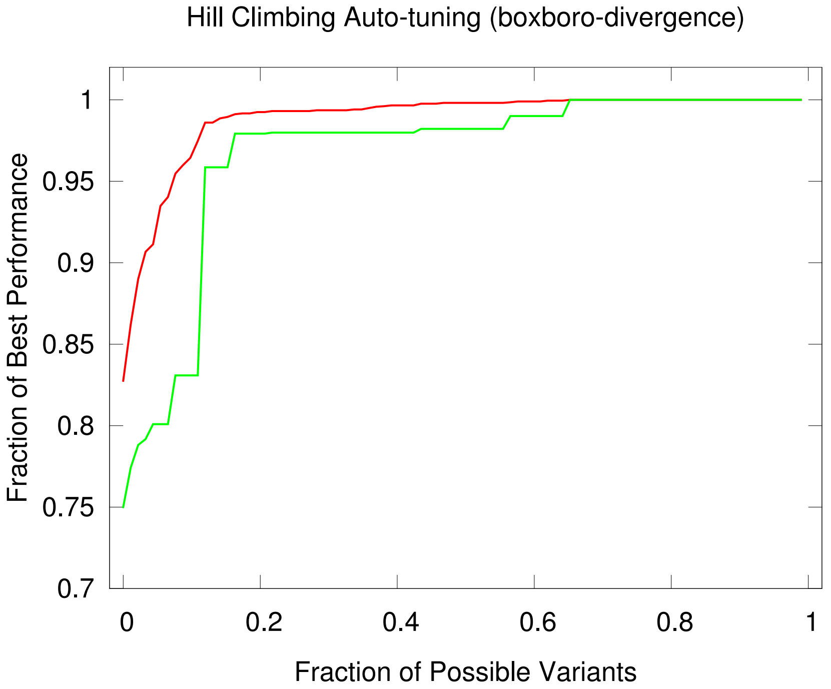
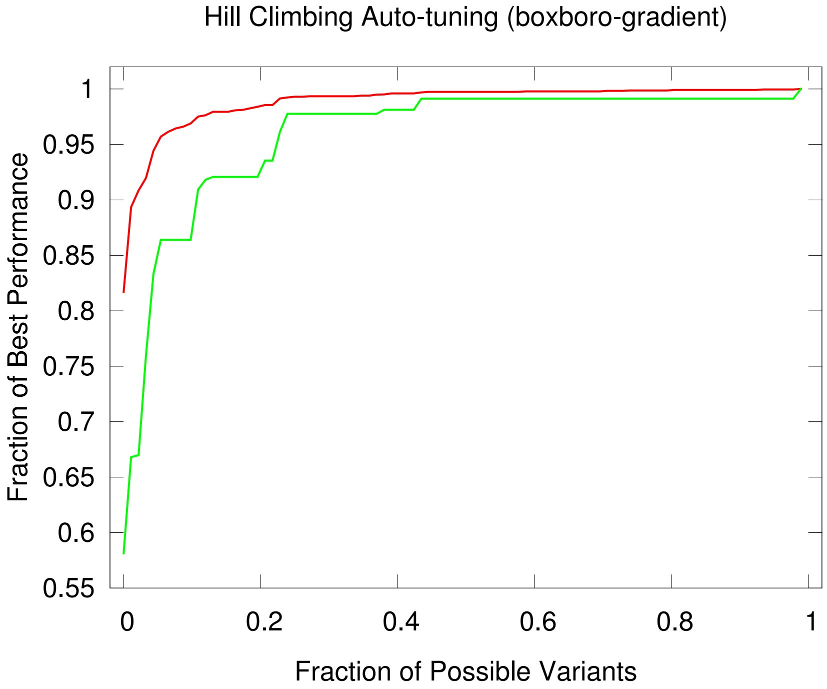


Figure 10.21: Hill climbing experiment on Boxboro, for 3D kernels running on a 256^3 grid for a single iteration. The top line shows the average of 25 runs of the experiment, while the bottom line shows the worst run.



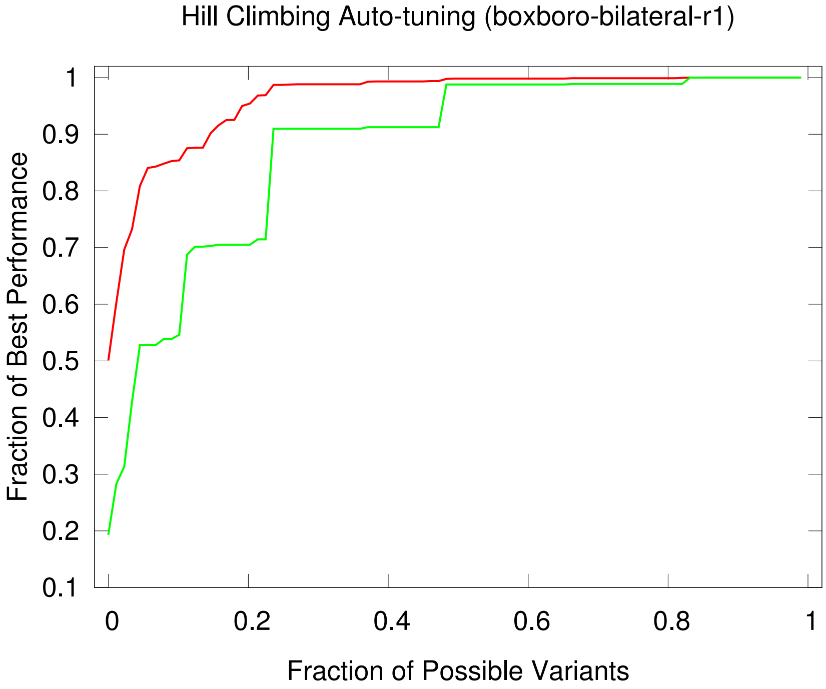


Figure 10.22: Hill climbing experiment on Boxboro (continued), for 3D kernels running on a 256^3 grid for a single iteration. The top line shows the average of 25 runs of the experiment, while the bottom line shows the worst run.

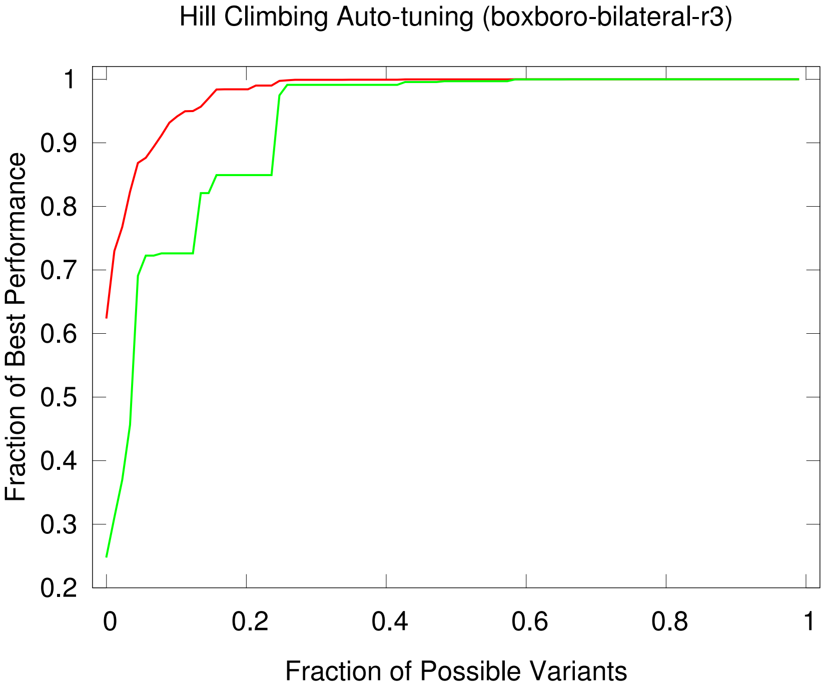


Figure 10.23: Hill climbing experiment on Boxboro (continued), for 3D kernels running on a 256^3 grid for a single iteration. The top line shows the average of 25 runs of the experiment, while the bottom line shows the worst run.

```
from stencil_kernel import *

class Prolongation2D(StencilKernel):
  def kernel(self, fine, coarse):
    for x in fine.interior_points(2,2):
      for y in self.neighbors(fine, x, 1):
        fine[y] = coarse[x]

class Restriction2D(StencilKernel):
  def kernel(self, coarse, fine):
    for x in coarse.interior_points():
      for y in self.neighbors(fine, x, 1):
        coarse[x] += 0.25 * fine[y]
```

Figure 10.24: Demonstration of using extensions to Sepya to express multigrid prolongation and restriction.

the extensions preserve our current restriction of disallowing right-hand-sides to access any points in the output grid except the point written to, the parallelization strategy can still exploit reordering the writes. We are also working on allowing Gauss-Seidel-style stencils in our language, where the input and output grids are the same. The traditional parallelization of such schemes involves dividing the points into separate sets that can be updated without dependencies (e.g. red-black); we are working on encoding this in our language to avoid the need for analysis.

10.5.2 Opportunities for Further Optimization

Unlike our proof-of-concept tuner, the current structured grid DSEL does not output GPU code. Future extensions to the compiler could generate code appropriate for GPUs, either using OpenCL or CUDA as the backend, both of which are supported by Asp.

Currently, boundary conditions are not optimized in our DSEL. Computations on boundary points occur in serial, and can incur a huge performance hit relative to the interior. Our code generator can be extended to generate boundary conditions in one of two ways: either the boundary calculation can be embedded into the grid traversal, which has the advantage of not incurring extra cache traffic, or they can be calculated in a separate parallel loop, which has the advantage of not require conditionals in the inner loop. Both approaches need to be explored to determine which is best.

Our auto-tuner and code generator only implements a subset of optimizations from [121]. For compute-bound kernels, it is important to implement others, including SIMDization and other optimization strategies that can improve the performance of such kernels.

10.6 Summary

This chapter presented Sepya, our structured grid DSEL and compiler, built using the Asp framework, which yields a high percentage of peak performance across a large number of structured grid kernels. The auto-tuning employed in the DSEL compiler is simpler than that in Chapter 9 yet still yields near-peak performance. The DSEL and compiler described here is more productive for performance experts to build, thanks to using the Asp infrastructure. In addition, users of Sepya can express their computations in few lines of code, yet still obtain over 80% of peak performance across machines and across kernels. The success of this DSEL points to the potential of the SEJITS approach for changing the paradigms of performance programming.

Chapter 11

Graph Algorithms

Graph algorithms are computations that operate on a *graph*, a mathematical abstraction consisting of a set of *vertices* as well as *edges* that connect pairs of vertices. This class of algorithms occurs frequently in a wide variety of domains, from within compilers to analyzing relationships between social networking users.

Graphs can be *undirected* or *directed*, meaning the edges in the graph have some notion of direction and thus distinguish between the two vertices being connected. For an edge in a directed graph, one vertex is the *source* and the other is called the *sink*. Figure 11.1 shows an example of a simple directed and undirected graph.

In this chapter, we examine some applications of graph algorithms as well as some basic building blocks for such algorithms in Section 11.1. Section 11.2 outlines some common representations of the graph data structure and related programming models. In Section 11.3 we describe the Knowledge Discovery Toolbox, the Python package for graph algorithms used as a basis for our DSEL compilers. In Section 11.4, we describe a Roofline model for graph computations that use the linear algebra representation, and Section 11.5 summarizes.

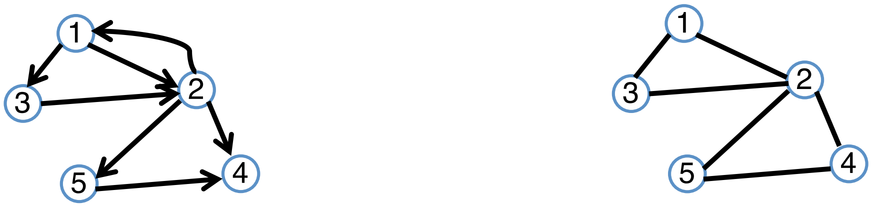


Figure 11.1: Example of a directed graph (left) and an undirected graph (right).

11.1 Applications of Graph Algorithms

Graph algorithms are used in a huge number of applications, spanning all domains of computer science. Indeed, the graph data structure is so essential it is often taught as part of basic computer science courses, and algorithms that operate on graph data structures are a massive topic of historical and current research. Analysis of graphs that arise from network connectivity, social networking relationships, search, and recommendation engines form a large part of modern usage of graph algorithms.

Many applications operate on standard directed or undirected graphs, but others use special types of graphs. These special graphs include directed acyclic graphs (DAGs, which are directed graphs with no cycles), trees (connected acyclic directed graphs where each vertex has a parent and zero or more child vertices), and hypergraphs (graphs in which many edges may exist between the same vertices). Such specialized graphs are building blocks for many applications; for example, compilers may use trees to represent program syntax internally.

Basic algorithms for graphs include Breadth First Search (BFS), Depth First Search (DFS), algorithms for finding minimum spanning trees, and shortest-paths algorithms. Breadth First Search and Depth First Search have similar structure, but result in different traversal order due to the former's use of a queue (ensuring vertices are processed in the order they are discovered) and the latter's use of a stack (which results in later vertices being processed before earlier ones). DFS and BFS are building blocks for many other important graph algorithms.

Depth First Search is a building block for topological sorts on DAGs and for finding strongly connected components (that is, maximal sets of vertices that, for each pair u, v both u is reachable from v and v is reachable from u). Breadth First Search is used to build a wide range of algorithms, including betweenness centrality (a measure of how important a vertex is to the structure of the graph) and maximal flow. It is also the benchmark algorithm used to see how well modern machines are at graph algorithms: the Graph 500 benchmark [28] uses BFS as its major component.

11.2 Common Programming Models

Graph algorithms use a variety of data representations and programming models, partially due to the wide variety of domains in which these algorithms are used. The classical data structure is an *adjacency list*, in which each vertex keeps a list of adjacent vertices (via pointers or indices into an array). A second common data structure is the *adjacency matrix*, where the graph is represented by a matrix, where the rows represent vertex id's for the source and columns represent vertex destinations (or vice-versa). Such matrices are symmetric if the graph is undirected.

Besides these canonical data structures, many others have been used in graph algorithm packages, depending on the requirements of application domain. Regardless of data structure, a variety of programming models exist for graph algorithms. In the next sections, we examine three common programming models and compare implementations of Breadth First Search in each.

11.2.1 Visitor Programming Pattern

In the graph visitor programming pattern, an algorithm is expressed in terms of a *visitor function* that is applied as each vertex is discovered. The input to the visitor is usually the discovered vertex. During each visit, new vertices to visit may be queued or directly visited.

A Breadth-First Search in this programming model consists of a visitor function that records, for each visited vertex, the "predecessor" vertex that added it to the visit queue. When no more unvisited vertices remain, the tree induced by the predecessor vertices represents the output BFS tree. Figure 11.2 shows Python pseudocode for the visitor in this programming pattern.

```
def visit(vertex, predecessor_id):
  vertex.predecessor_id = predecessor_id
  visit_queue.extend(vertex.neighbors)
```

Figure 11.2: Visitor function for Breadth First Search. The outermost loop is not shown, but usually handles ensuring a vertex is only visited once as well as visiting vertices in the queue in order.

```
def vertex_func(v):
  if v.messages:
    for n in v.neighbors:
      v.send_message(n, v.id)
    v.predecessor = v.messages[0] # choose first
    v.finish()
```

Figure 11.3: Vertex function for Breadth First Search. The outermost loop is not shown, but usually executes the function for each non-finished vertex in the graph and handles communication between supersteps.

11.2.2 Bulk-Synchronous Programming Model for Graph Algorithms

In the Bulk-Synchronous Programming Model (BSP [114]), typified by Google's Pregel [75] package and the GraphLab [73] package, users write a per-vertex function that runs at each *superstep* until a global agreement is reached that the algorithm is finished. In each superstep, vertices can send messages to other vertices or declare themselves finished. A message sent during one superstep is available for the target vertex to process at the next superstep.

In this programming model, the vertex function simply sends messages to its adjacent vertices to set their predecessor to the vertex. In addition, each vertex chooses one of its incoming messages to be its predecessor and then signals that it is finished. After enough supersteps, each vertex has recorded its predecessor vertex and the resulting tree represents the result of BFS. A pseudocode version of a vertex function for BFS is shown in Figure 11.3.

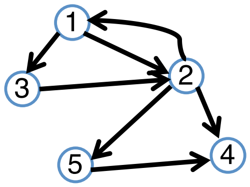
11.2.3 Matrix Representation & the Linear Algebra Programming Model

The linear algebra programming model for graph algorithms treats the graph as a (sparse) matrix and conceptualizes graph algorithms as sequences of linear algebra operations (such as sparse matrix-vector multiply) using algorithm-specific *semiring operations*. Recall that a semiring is an algebraic structure that consists of a set and two operations: addition and multiplication, where addition is a commutative monoid and multiplication is a (possibly non-commutative) monoid. In addition, multiplication has a zero element that is the same as addition's identity element. In contrast to a ring, semirings do not necessarily have an additive inverse.

Writing a graph algorithm in this programming model involves defining both the sequence of linear algebra operations as well as the semiring operations used for each kernel application. Many graph algorithms are amenable to being written in this manner, including Breadth First Search, Betweenness Centrality, Maximal Independent Sets, and Shortest Paths (Bellman-Ford) [62].

The advantage of this approach is that casting graph algorithms as linear algebra operations enables taking advantage of decades of optimization and tuning work that has been applied to the linear algebra domain. Many libraries with a rich history exist for performing sparse linear algebra operations on multicore and distributed computers (such as LAPACK [3] and ScaLAPACK [15]), and extensive research in this area has explored a large optimization space. Potentially, high performance graph algorithms can be written by leveraging this existing work with slight modifications for custom semiring operations.

Breadth First Search in the Linear Algebra Programming Model



$$\begin{array}{ccccc} & & A & & \\ . & 1 & . & . & . \\ 1 & . & 1 & . & . \\ 1 & . & . & . & . \\ . & 1 & . & . & 1 \\ . & 1 & . & . & . \end{array} \times \begin{array}{c} x \\ 0 \\ 0 \\ 3 \\ 0 \\ 0 \end{array} = \begin{array}{c} b \\ 0 \\ 3 \\ 0 \\ 0 \\ 0 \end{array} \qquad \begin{array}{c} \text{parents} \\ 0 \\ 3 \\ . \\ 0 \\ 0 \end{array}$$

$$\begin{array}{ccccc} & & A & & \\ . & 1 & . & . & . \\ 1 & . & 1 & . & . \\ 1 & . & . & . & . \\ . & 1 & . & . & 1 \\ . & 1 & . & . & . \end{array} \times \begin{array}{c} x \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \end{array} = \begin{array}{c} b \\ 2 \\ 0 \\ 0 \\ 2 \\ 2 \end{array} \qquad \begin{array}{c} \text{parents} \\ 2 \\ 3 \\ . \\ 2 \\ 2 \end{array}$$

Figure 11.4: Breadth First Search using linear algebra on a simple graph. The example shows BFS starting from vertex 3; in this case, the search is finished after two steps.

To demonstrate how graph algorithms can be performed by composing linear algebra operations, we will walk through an example of Breadth First Search on a simple graph. Figure 11.4 shows a small graph, on which we will step through a BFS starting with vertex 3. The major operation in the BFS is sparse matrix vector multiply (SpMV), with a special semiring that, for addition returns the maximum input and for multiplication, simply returns the second input. In other words, $a + b = max(a, b)$ and $a \times b = b$. In addition, null entries in the matrix are not used in the SpMV, so the semiring is not applied to them.

The algorithm begins by setting the x vector such that the start vertex's corresponding entry is set to its id. In this case, we set the entry in the third row to 3. Note that the matrix we use is the transpose of the traditional adjacency matrix. After the first SpMV, the result vector has a value of 3 for every vertex discovered in this first iteration; in this case, the only vertex discovered is 2. The result is merged with the parents vector, which is the output of the algorithm; essentially, the vector-vector operation here replaces the zero entries in the parents vector if the entry is nonzero in the b vector.

At the second step (Figure 11.4, bottom), we set the entries corresponding to the newly-discovered vertices to their ids (as we did with the initial vertex). Thus, we set the id for vertex 2 in the x vector. After the second SpMV, all the vertices that are reachable from vertex 2 have an entry of 2 in the result vector. Once again, we update the parents vector with this result. In this example, that leads to all vertices being discovered, which ends the algorithm.

This simple example demonstrates how graph algorithms can be decomposed into linear algebra operations. In BFS, we primarily use SpMV and a vector-vector operation, but other algorithms can use matrix-matrix multiplication as well as other linear algebra primitives.

11.3 KDT: The Knowledge Discovery Toolbox

The Knowledge Discovery Toolbox (KDT) [74] is a Python package for high performance parallel computations on graphs, designed for domain experts who are not computer scientists. Using the package, scientists do not need to understand the underlying data structures or complex algorithms, but can perform computations at a high level. The goal is for users of KDT to never need to write any low-level code, and instead, perform all their computations in Python.

KDT is built on the Combinatorial BLAS [20], which is a C++ library for graph computations using linear algebra representations for the graph and algorithms. Parallelism in the Combinatorial BLAS uses the MPI library. CombBLAS includes structures for graphs as sparse matrices as well as a small set of primitives that allow algorithms to be expressed as computations on matrices and vectors using specialized algorithm-specific semirings.

Because KDT is not limited to small-world graphs, it targets computation on both single-node highly-parallel shared memory systems as well as large, distributed clusters. Thus, much of KDT is written as low-level C++ functionality and interfaces on top of the Combinatorial BLAS building blocks for use from Python. Interfacing between Python and C++ is done using the SWIG [10] framework.

The built-in graph algorithms in KDT are written in a mix of Python (for productivity reasons) and C++ (for efficiency reasons). For a number of graph algorithms, the semiring operations used are written in C++ since they are used extensively as building blocks in KDT. Furthermore, these operations are type-specialized to operate with the included vertex and edge types. Currently, KDT requires users to declare custom vertex and edge types in C++ , while limiting them to only two types. In addition to the algorithms themselves, KDT contains functionality that allows users to selectively apply the algorithms to a subset of the graph by specifying filters in Python that dictate whether a given input edge or vertex is to be included. More on this functionality is covered in the next chapter.

Because the algorithms that use Python in their innermost operations require serialized calls into the interpreter, the performance can be much lower than desired. The SEJITS approach can be used to mitigate these slowdowns. The domain-specific embedded languages for KDT in the next chapter interject translated/compiled C++ code into the Python-C++ internal KDT interface.

11.4 Performance Modeling Issues for Graph Algorithms Using Linear Algebra

Roofline models for graph algorithms are highly dependent on the data structure and algorithm under question. In this section, we summarize a Roofline model for Breadth First Search in KDT and the Combinatorial BLAS, which was derived in [19].

For characterizing operational performance, the methodology is to simply run a BFS on a graph that is small enough to fit into the processor caches of the machine under study; this is run a large number of times to amortize any initial costs due to traffic from main memory. With this test, the in-core performance limit is found.

Characterizing memory bandwidth requirements is somewhat more complicated. Memory access patterns during BFS can be characterized into three regimes:

1. *Streaming access*, which is defined as accesses that occur to consecutive memory locations. These result from accessing vertex pointers as well as when creating a vector for each frontier before performing the sparse matrix vector multiply.

2. *Stanza-like access*, in which a few accesses to consecutive locations occur, followed by a jump in access location. These occur due to accesses into the adjacency list; these are essentially reads of some set stanza size, with subsequent stanzas being to completely different locations in memory.

3. *Random access* due to updating the list of visited vertices and for accessing the edge data structure from the graph structure. These accesses look like random accesses to locations in memory, with no spatial or temporal locality.

For each of the access types, a modified STREAM benchmark can be written that characterizes the memory bandwidth performance of each access type; based on this, a limit for performance due to memory bandwidth requirements can be determined for a particular graph. The addition of edge filtering changes the mix of kinds of access as well, altering the memory bandwidth limit.

Because the Roofline model is complex and highly data-dependent, we will generally compare performance to a known baseline: graph algorithms using the C++ implementation inside KDT and the Combinatorial BLAS. For a characterization of the filtering DSEL that compares against this derived roofline limit, see [19].

11.5 Summary

Graph algorithms are in important class of computation for many areas of computer science, and, due to increasing graph sizes, require high performance in order to be useful. Several programming models exist for such algorithms, one of which is to cast them as linear algebra, an approach used by KDT and the Combinatorial BLAS. In the next chapter, we build DSELs to accelerate KDT while providing high-level programmability to the user.

Chapter 12

Domain Specific Embedded Languages For High Performance Graph Algorithms in the Knowledge Discovery Toolbox

In this chapter, we build two domain-specific embedded languages (DSELs) for the Knowledge Discovery Toolbox graph algorithms package. Leveraging this existing framework for large-scale graph algorithms in Python, we demonstrate how the SEJITS approach can be applied to existing high-level software packages to mitigate slowdowns caused by using the high-level language. The performance goals of the DSELs in this chapter are to match the performance of writing graph operations in low-level languages, while preserving the productivity of writing in a high-level language.

In Section 12.1, we build a domain-specific language for filtering semantic graphs when applying existing graph algorithms. This allows users of KDT to selectively apply algorithms on graphs with information on the edges. The DSEL is described in Section 12.1.2 and the performance of our approach is evaluated in Section 12.1.3. Then, in Section 12.2 we build a DSEL for defining semiring operations, enabling new algorithms to be developed for KDT without the need to write them in C++ for performance. Section 12.3 outlines some next steps to extend the DSEL functionality demonstrated in this chapter, and the chapter is summarized in Section 12.4.

12.1 A Domain-Specific Embedded Language for Filtering Semantic Graphs

In large scale graph analytics, graphs vertices usually represent entities of interest while the edges connecting them represent specific kinds of relationships. Such graphs are often referred to as *semantic graphs*. In many cases, the user of a graph analytics package is interested in running high-level algorithms such as betweenness centrality or reachability analysis on the subset of the graph that denotes a specific kind of relationship or a specific subset of the entities, while ignoring the other edges or vertices.

As a running example, consider a graph that maps Twitter relationships. In this graph, nodes represent users and edges represent relationships; what kind of relationship is encoded in the edge type. It is useful to use this graph to do queries, some of which care about certain edge types

```
class MyFilter(object):
 def __call__(self, e):
  # if it is a retweet edge
  return (e.count > 0 and
      # and it is before June 30
      e.latest < strtoftime("2009-6-30")
```

Figure 12.1: Example of an edge filter in KDT. The filter object implements Python's callable convention.

only, without having to instantiate a new graph with other edges pruned. Such operations can be performed either by pre-pruning the unwanted vertices and edges (and therefore creating a new graph) before running the algorithms, or by *filtering* the graph on-the-fly— that is, checking each vertex or edge before using it within the algorithm.

In this section, we build a Domain-Specific Embedded Language for on-the-fly filtering of semantic graphs. The end result allows users to write Python filters expressing the edges they would like to include without incurring large performance penalties due to the per-edge or per-vertex check. Such filters can be applied to any algorithm in KDT in a transparent manner, by piggybacking on existing KDT functionality.

12.1.1 Filters in the Knowledge Discovery Toolbox

In KDT, filters are expressed as Python *callable* objects. In most cases, this is either an instance of a class that has a `__call__()` method, or is a Python lambda. An example of a filter is shown in Figure 12.1.

The implementation of on-the-fly KDT filters works by applying the filter to each edge right before calling the semiring operation on the edge, using an upcall into Python to run the filter function. If the filter returns false, the semiring operation returns the semiring's additive identity (SAID), which the underlying CombBLAS operations then use as a trigger to not store the data item in the result. This is because CombBLAS uses sparse data structures in which the SAID should never be stored (similar to not storing zeros in usual sparse matrix formats such as Compressed Sparse Row). In this way, the end result is as if the operation never occurred, both in terms of the result and in terms of the returned data structure. The major drawback here is that the upcall into Python is very expensive and occurs once for every single edge.

Materializing a filtered matrix in KDT means instantiating a matrix that has been pruned of edges that do not pass the filter. Operationally, materializing just applies the filter to each edge of a matrix before performing any operations, and a new matrix is constructed, made up only of edges that pass the filter, while the original matrix is preserved. The major drawback of this is that creating a large materialized graph can take a very long time and in fact may be impossible for very large graphs due to memory size constraints. The upside is that once it is created, subsequent operations no longer need to incur the overhead of an upcall per edge.

From the user's perspective, most if not all of these implementation details are hidden. All they need to do is write a filter and add it to the matrix. However, the performance hit is more than large enough that users will notice the slowdown when filtering, especially if they use KDT algorithms that have been implemented partially in C++ for speed.

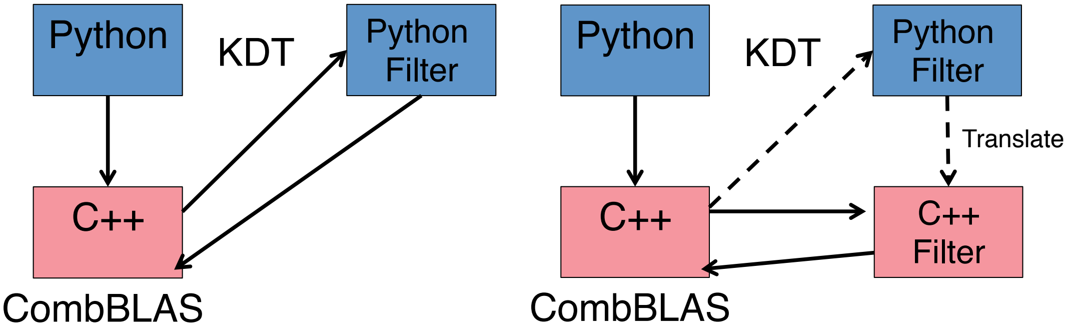


Figure 12.2: Left: Calling process for filters in KDT. For each edge, the C++ infrastructure must upcall into Python to apply the filter. Right: Using our DSL for filters, the C++ infrastructure calls the translated version for each edge, eliminating the upcall overhead.

12.1.2 DSEL for Filters

By defining an embedded DSL for KDT filters, and then translating it to C++ , we can avoid performance penalties while still allowing users the flexibility to specify filters in Python. In this manner, the upcall to Python will be eliminated, and filtering will occur at the C++ level. We use the Asp framework to implement our DSEL.

Our approach is shown in Figure 12.2. In the usual KDT case, filters are written as simple Python functions. Since KDT uses the Combinatorial BLAS at the low level to perform graph operations, each operation at the Combinatorial BLAS level must check to see whether the vertex or edge should be taken into account, requiring a per-vertex or per-edge upcall into Python. Furthermore, since Python is not thread-safe, this essentially serializes the computation in each MPI process. Though the Combinatorial BLAS currently does not use shared memory parallelism, this serialization prevents KDT from benefiting from such optimizations should they be implemented.

In this section, we define an embedded domain-specific language for filters, and allow users to write their filters in this DSEL, expressed as a subset of Python with normal Python syntax. Then, at instantiation, the filter source code is introspected to get the Abstract Syntax Tree (AST), converted to an intermediate form, and then translated into low-level C++ . Subsequent applications of the filter use this low-level implementation, sidestepping the serialization and cost of upcalling into Python.

Now we define our domain-specific language and show several examples of filters written in Python.

Semantic Model for Filters

Recall that in our approach, we first define the *Semantic Model* of filters, which is the intermediate form of our DSEL. The Semantic Model expresses the semantics of filters, though its definition looks similar to a syntax definition. After defining this, we then map pure-Python constructs to constructs in the Semantic Model. It is this pure-Python mapping that users use to write their filters, and it is instances of the Semantic Model that our backend code generators use to generate C++ source.

In defining the Semantic Model, we must look at what kinds of operations filters perform. In particular, vertex and edge filters are functions that take in one or two inputs and return a boolean value. Within the functions, filters must allow users to inspect fields of the input data types, do comparisons, and perhaps perform arithmetic operations with data fields. In addition, we want to (as much as possible) prevent users from writing filters that do not conform to our assumptions, have side effects, or otherwise are incorrect; although we could use analysis for this, it is much simpler to construct the language in a manner that prevents users from writing non-conformant filters. If the filter does not fit into our language, we run it in the usual fashion, by doing upcalls into pure Python. Thus, if the user writes their filters correctly, they achieve fast performance, and otherwise the user experience is no worse than before— the filter still runs, just not at fast speed.

The Semantic Model is shown in Figure 12.3. We have built this to make it easy to write filters that are "correct-by-construction;" that is, if they fit into the Semantic Model, they follow the restrictions of what can be translated. For example, we require that the return be provably a boolean (by forcing the BoolReturn node to have a boolean body), and that there is either a single input or two inputs (either UnaryPredicate or BinaryPredicate). These restrictions make code generation straightforward.

Given the Semantic Model, now we define a mapping from Python syntax to the Semantic Model.

Python Syntax for the Filter DSEL

Users of KDT are not exposed to the Semantic Model. Instead, the language they use to express filters in our DSEL is a subset of Python, corresponding to the supported operations. Informally, we specify the language by expressing a limited API that conveys what a filter must do and what operations can be inside a filter: namely, a filter takes in one or two inputs (that are of pre-defined edge/vertex types), must return a boolean, and is allowed to do comparisons, accesses, and arithmetic on immediate values and edge/filter instance variables. In addition, to facilitate translation, we require that a filter be an object that inherits from the `PcbFilter` Python class, and that the filter function itself is a member function called `__call__`. This requirement of using a specific superclass is one of the characteristics of DSELs written using Asp. From the perspective of a KDT user, this looks similar to a standard Python API description.

The example KDT filter from Figure 12.1 is presented in SEJITS syntax in Figure 12.4. Note that because a filter cannot call a function, we must use an instance variable to compare against the timestamp; this instance variable, however, must be finalized before the filter call and cannot be set by the filter. Even given our relatively restricted syntax and semantics, users can specify a large class of useful filters in our DSEL. In addition, if the filter does not fit into our DSEL, it is still executed using the slower upcalls to pure Python after issuing a warning to the user.

Interfacing with KDT

We modify the underlying C++ filter objects used by KDT's Python/C++ bridge, which are instantiated with pointers to Python functions, by adding a function pointer that is checked before executing the upcall to Python. This function pointer is set by our translation machinery to point to the translated function in C++ . When executing a filter, the pointer is first checked, and if non-null, directly calls the appropriate function.

Compared to Combinatorial BLAS, at runtime we have the additional sources of overheads relating to the null check and function pointer call. In addition, because this function call occurs

```
# top-level node for a filter
UnaryPredicate(input=Identifier, body=BoolExpr)

BinaryPredicate(inputs=Identifier*, body=BoolExpr)
    check assert len(self.inputs)==2

Expr =  Constant
          | Identifier
          | BinaryOp
          | BoolExpr

Identifier(name=types.StringType)

BoolExpr = BoolConstant
          | IfExp
          | Attribute
          | BoolReturn
          | Compare
          | BoolOp

Compare(left=Expr, op=(ast.Eq | ast.NotEq | ast.Lt | ast.LtE | ast.Gt | ast.GtE), right=Expr)

# this is for multiple boolean expressions
BoolOp(op=(ast.And | ast.Or | ast.Not), operands=BoolExpr*)
    check assert len(self.operands)<=2

Constant(value = types.IntType | types.FloatType)

BinaryOp(left=Expr, op=(ast.Add | ast.Sub), right=Expr)

BoolConstant(value = types.BooleanType)

IfExp(test=BoolExpr, body=BoolExpr, orelse=BoolExpr)

# this if for a.b
Attribute(value=Identifier, attr=Identifier)

# the return value must provably be a boolean
BoolReturn(value = BoolExpr)
```

Figure 12.3: Semantic Model for KDT filters using SEJITS.

```
class MyFilter(PcbFilter):
  def __init__(self, target_date):
    self.target = strtoftime(target_date)

  def __call__(self, e):
    # if it is a retweet edge
    if (e.count > 0 and
        # and it is before the target date
        e.latest < self.target):
      return True
    else:
      return False
```

Figure 12.4: Example of an edge filter that the translation system can convert from Python into fast C++ code.

	First Run	Subsequent
Codegen	545 ms	-
Compile	4210 ms	-
Import	32 ms	32 ms

Table 12.1: Overheads of using the filtering DSEL.

via a pointer into a dynamically loaded library, it incurs a higher overhead than a normal function call. However, relative to the non-translated KDT machinery, these are trivial costs for filtering, particularly compared to the penalty of upcalling into Python.

Overheads of code generation are shown in Table 12.1. On first running using a particular filter, the DSEL infrastructure translates and compiles the filter in C++ ; most of the time here is spent calling the external C++ compiler, which is not optimized for speed. Subsequent calls only incur the penalty of Python's `import` statement, which loads the cached library.

12.1.3 Experimental Results

Before presenting our experiment, it is important to characterize the size of the DSEL compiler implementation, especially relative to the size of KDT. Table 12.2 shows the lines of code for KDT and our filter DSEL. The DSEL implementation is quite small, consisting of small modifications to KDT at the C++ level and a DSEL compiler implemented in Asp.

To test the behavior of SEJITS-enabled filtering in Python, we compare against using the Combinatorial BLAS directly (that is, writing custom semirings in C++) and against using KDT's default filtering mechanism.

	Python LOC	C++ LOC
KDT	7009	8177
Filter DSEL	194	24

Table 12.2: Lines of Code for KDT and the Filter DSEL. Code generated by SWIG is elided from the KDT line counts. Counts are generated using the CLOC tool [2].

```
struct TwitterEdge {
  bool follower;
  time_t latest; // set if count > 0
  short count; // number of retweets
};
```

Figure 12.5: The C++ data structure for edges in these experiments. The edge type is encoded by either setting `follower` true (in which case it is a following relationship) or by having `count` and `latest` set, corresponding to the number of retweets and when the latest one occurred.

Parameter	Value
Edge Factor	16
a	0.59
b	0.19
c	0.19
d	0.05

Table 12.3: R-MAT parameters used in this study. Note that an R-MAT graph of a particular scale N has 2^N vertices and approximately $EdgeFactor \times 2^N$ edges.

The edge data structure used corresponds to our Twitter example, and is shown in Figure 12.5. Edges in this graph encode the relationship between Twitter users; the two types of relationships are following and retweeting. We use both real graphs from anonymized Twitter data as well as synthetic R-MAT [72] matrices.

The R-MAT matrices are generated by KDT's built-in R-MAT generator using parameters shown in Table 12.3. Once a boolean matrix of a specific scale is generated, edge types are set using a random number generator, and the data structure is converted to use our Twitter edge data type. Edge types are generated to guarantee a particular filter permeability by weighting the random number generator.

In addition to the synthetic data, we use graphs from anonymized Twitter data in which edges represent following and retweeting relationships. Specifically, an edge from v_i to v_j where `follower` is set to true encodes user i follows user j. If there is an edge from v_i to v_j with `count` greater than zero, the edge encodes the fact that user i has retweeted user j `count` times, with the latest date recorded. The real data represents interactions that occurred in the period June to December

	Vertices		Edges	
		Tweet	Follow	Tweet&Follow
Small	0.5	0.48	65.3	0.3
Medium	4.2	14.2	386.5	4.8
Large	11.3	59.7	589.1	12.5
Huge	16.8	102.4	634.2	15.6

Table 12.4: Sizes (vertex and edge counts) in millions of different combined Twitter graphs. Data courtesy Aydın Buluç.

2009. In order to use the data for scaling studies, we use subsets of the data based on time windows. Statistics for the four subsets used are shown in Table 12.1.3. For the huge dataset, the amount of memory required is dozens of GB for the graph alone, not including auxiliary data structures, vectors, or MPI buffers. Unlike the synthetic data, the real Twitter data is directed, and we only report numbers for runs of BFS that reach the largest strongly-connected components of the graphs, in order to ensure the BFS touches a large portion of the graph.

Our performance goal is to obtain performance for the DSEL that is as close as possible to the pure Combinatorial BLAS approach, but we expect some overheads. First-run overheads (shown in Table 12.1) are elided in our performance evaluation, since they only occur the first time a filter is instantiated. However, the DSEL approach does incur additional overheads relative to the Combinatorial BLAS: there are two additional function calls for the filters, and these function calls are to dynamically-loaded libraries, which result in approximately two extra cycles per call versus function calls within the same module on the 64-bit x86 architecture. Since the function bodies are very small, and the calls occur for every single edge in the graph, we expect this overhead to be non-trivial.

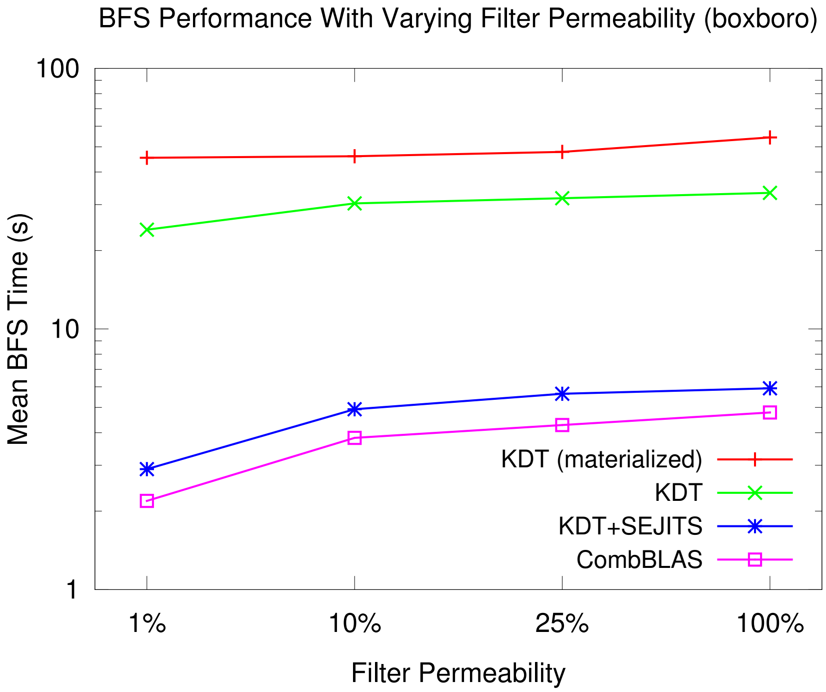
Furthermore, on Hopper dynamic linking is not well-supported, resulting in very large overheads for the first calls into a dynamic library as well as large performance variability due to loading these libraries from a shared file system.

Figure 12.6 shows the performance of filtered Breadth-First Search on Boxboro and Hopper for our synthetic runs as filter permeability is changed. On Boxboro, KDT with our filter DSEL consistently outperforms pure Python filtering by $5 - -6\times$, greatly reducing the overhead of filtering. Compared to the pure Combinatorial BLAS approach, our filtering DSEL is at most only 20% slower. On Hopper, the DSEL performance is about $5\times$ faster than KDT, but is slower than Combinatorial BLAS performance by up to a factor of $2.6\times$. Much of this slowdown is due to poor dynamic loading performance on Hopper, but is also related to inter-compute node variability on the large scale machine.

Filtered BFS performance on the real data sets is shown in Figure 12.7. On both Boxboro and Hopper, the DSEL implementation closely tracks the performance of writing the filter into a semiring at the Combinatorial BLAS level; the performance is indistinguishable between the two. Compared to the current KDT filtering mechanism, the SEJITS approach is $4 - 5\times$ faster on both machines. Thus, on real data, all of the overhead of using Python is mitigated.

Strong scaling performance for filtered Breadth-First Search on Boxboro is shown in Figure 12.8. The Combinatorial BLAS shows a remarkable scaling performance of $35\times$ on 36 processors, closely matched by BFS using our filtering DSEL, which obtains $34\times$. KDT with Python filters obtains less than $30\times$ on 36 processors, which is still a good scaling number given the overheads inherent in the implementation. The high level of parallel efficiency for KDT with SEJITS demonstrates how our DSEL eliminates the overheads of using Python for filtering.

Strong scaling performance for filtered BFS on Hopper is shown in Figure 12.9. Again, the Combinatorial BLAS implementation shows almost perfect scaling up to 1024 processors, but the DSEL performance shows some variability due to limitations of Hopper. Nevertheless, in most cases the strong scaling is relatively good and tracks the Combinatorial BLAS implementation.



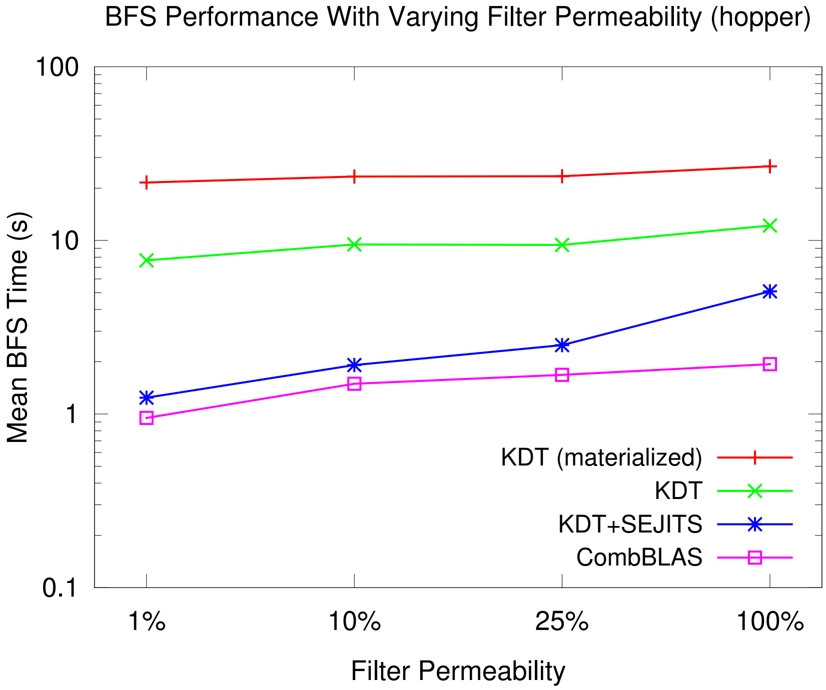


Figure 12.6: BFS performance as filter permeability is changed. For Boxboro, the graph is R-MAT scale 23 running on 36 cores, and for Hopper it is scale 25 running on 576 processors.

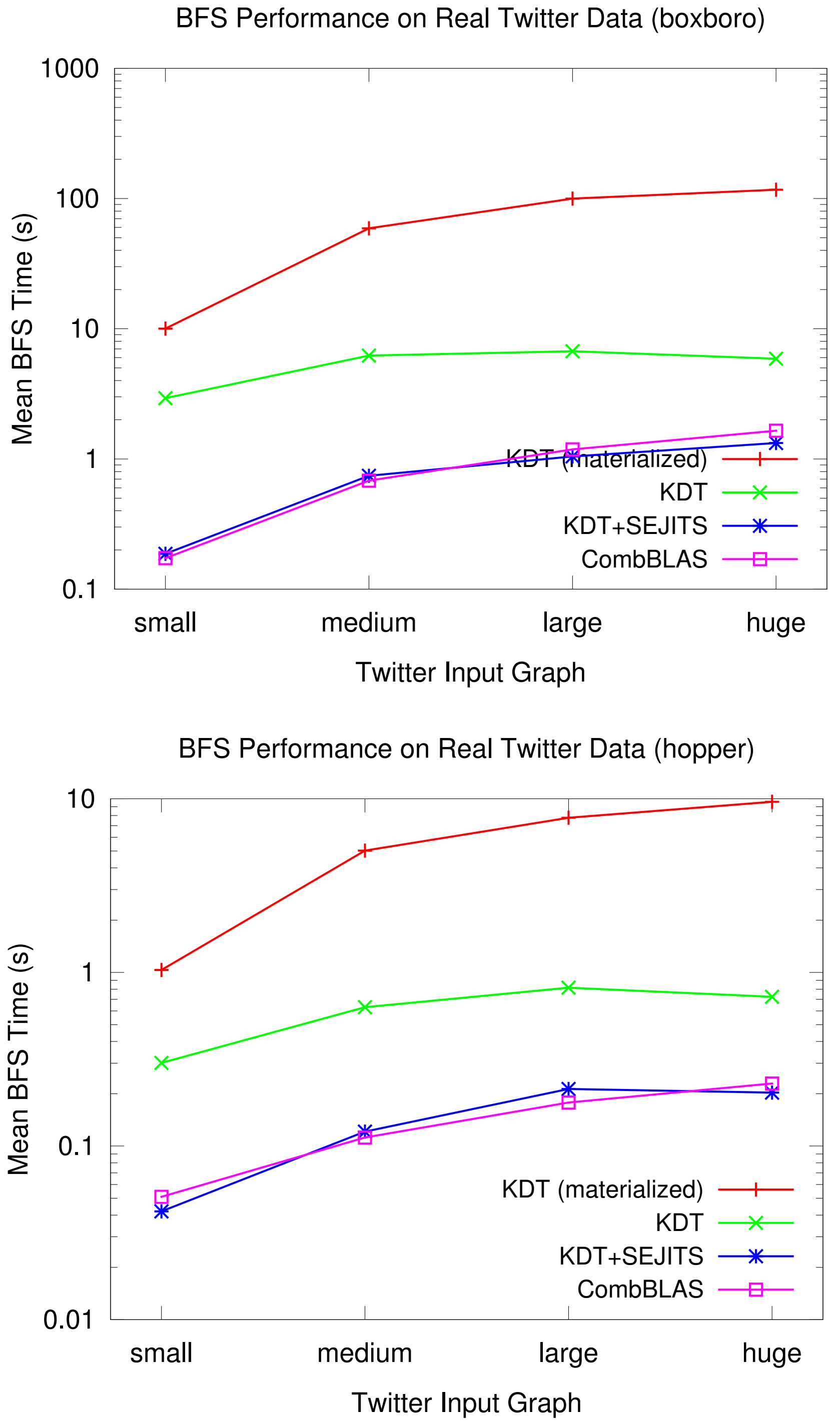


Figure 12.7: Filtered BFS performance for the real Twitter datasets. For Boxboro, the runs use 36 cores, and for Hopper they use 576 cores.

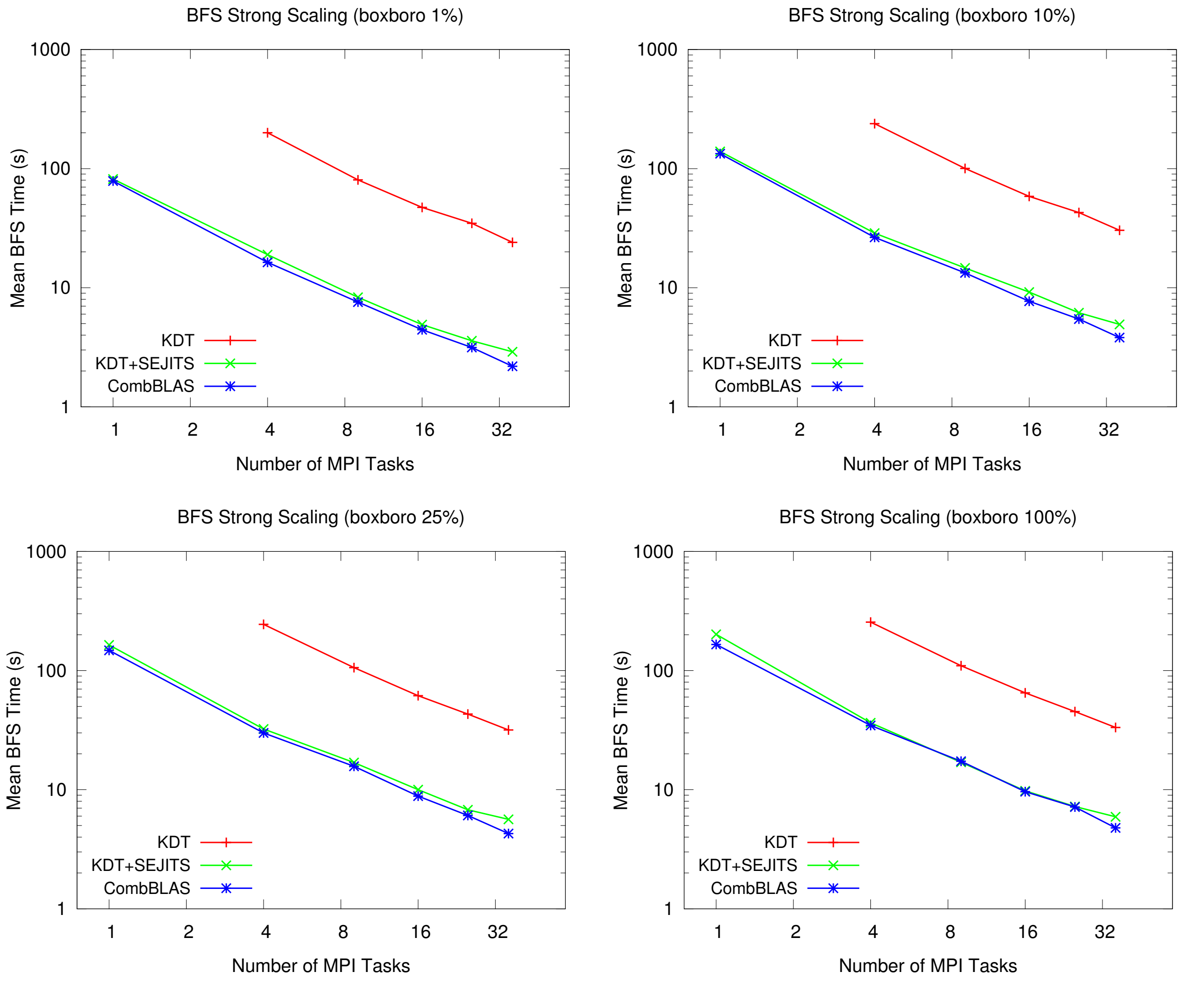


Figure 12.8: BFS strong scaling performance as filter permeability is changed on Boxboro for synthetic R-MAT matrices of scale 23. Note that runs are done with perfect square numbers of processors due to restrictions in KDT.

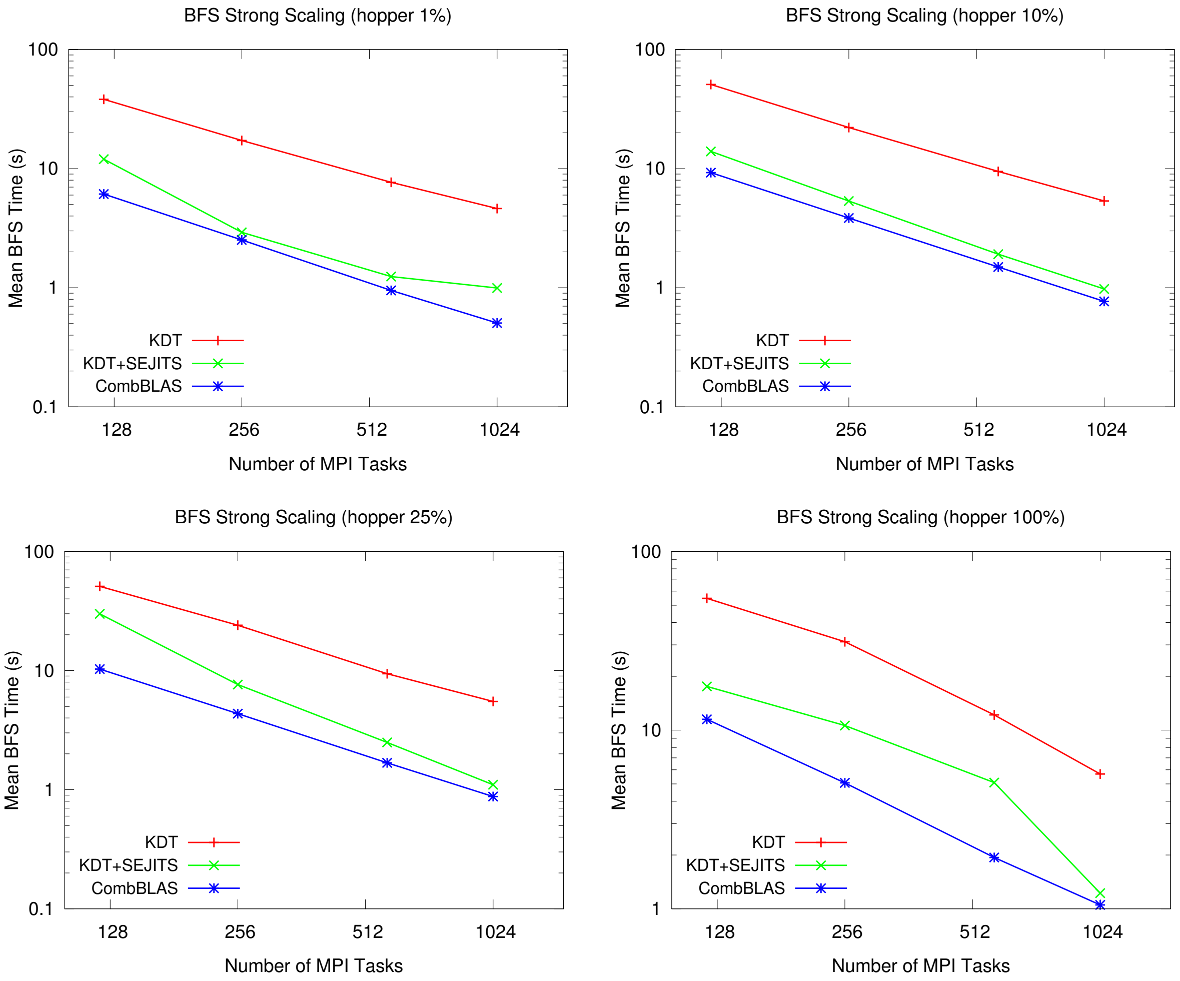


Figure 12.9: BFS strong scaling performance as filter permeability is changed on Hopper for synthetic R-MAT matrices of scale 25. Note that runs are done with perfect square numbers of processors due to restrictions in KDT.

12.2 A Domain-Specific Embedded Language for Defining Semirings in Python

As described in Chapter 11, graph algorithms on graphs that use the linear algebra representation consist mostly of standard linear algebra operations such as sparse matrix vector multiply (SpMV) and vector-vector operations along with definitions for the two binary operations in the semiring. Implementing different algorithms requires thinking about the algorithm in terms of these matrix operations as well as defining the custom semiring operations for addition and multiplication.

Currently, in KDT these operations can be either defined as part of KDT's interface with the Combinatorial BLAS (in other words, in C++) or using Python callables (functions, lambdas, or classes that implement a `__call__` method with the correct arity). In this section, we extend KDT's capabilities by introducing a second DSEL for such operations, allowing algorithm developers to write their semiring operations in Python but mitigating the problems with the existing pure Python approach, particularly the poor performance due to upcalls to Python.

12.2.1 Semirings in KDT

In KDT, semiring operations are used by matrix-vector and matrix-matrix operations, although the same underlying abstractions are used for vector-vector operations. In particular, semirings and vector operations use instances of the `UnaryFunction` or `BinaryFunction` C++ classes, which can be called by SpMV or other C++ operations both in the Combinatorial BLAS as well as in KDT's C++ layer. The `BinaryFunction` particular class takes in two inputs (which can be any of KDT's built-in types) and returns a single output; in most cases, this output is of the same class as one of the inputs. The `BinaryFunction` class is partially templated to deal with its polymorphic nature. `UnaryFunction` is the analog for unary functions used in low-level operations in KDT.

In the implementation, Python callbacks are used to implement the function itself, and are wrapped and called using SWIG. The inputs and outputs are casted to/from Python objects. Alternatively, some commonly-used operations are implemented in C++ for efficiency. For example, the semiring used in Breadth First Search (BFS) is the `(min, select2nd)` semiring, which returns the minimum input for the add and for multiply, returns the second input (see Chapter 11 for an example of BFS that uses a similar semiring), is defined in C++ since it is used as a building block for many algorithms in KDT.

12.2.2 Domain-Specific Embedded Language for Semiring Operations

To define an appropriately-restricted DSEL for `UnaryFunction`/`BinaryFunction` objects that are used in semiring operations, we first characterize some of the common patterns used for such functions in KDT. The most commonly-used semiring operations simply return one of their inputs, but another common pattern is to look at some property of the inputs (such as whether they are equal to a number) and, based on this, either return one of the inputs or an immediate value. Finally, some semiring operations perform the usual arithmetic operations, sometimes after querying properties of the inputs.

Based on these common patterns, we define our DSEL using the intermediate representation in Figure 12.10. The Semantic Model is surprisingly similar to the one used in our filtering DSEL describe previously; this allows us to reuse much of the transformation infrastructure built for that DSEL. However, note that we do not require a specific type of return statement like we do in the filter DSEL.

The basic building blocks in the Semantic Model are enough to express the semiring operations used in most of the algorithms included in KDT, including Breadth-First Search, Betweenness Centrality, and PageRank [90]. Further functionality can be added if needed by future algorithms implemented in KDT.

```
UnaryFunction(input=Identifier, body=Expr)

BinaryFunction(inputs=Identifier*, body=Expr)

Expr =  Constant
          | Identifier
          | BinaryOp
          | BoolConstant
          | IfExp
          | Attribute
          | FunctionReturn
          | Compare

Identifier(name=types.StringType)

Compare(left=Expr, op=(ast.Eq | ast.NotEq | ast.Lt | ast.LtE | ast.Gt
| ast.GtE), right=Expr)

Constant(value = types.IntType | types.FloatType)

BinaryOp(left=Expr, op=(ast.Add | ast.Sub | ast.And), right=Expr)

BoolConstant(value = types.BooleanType)

IfExp(test=(Compare|Attribute|Identifier|BoolConstant|BinaryOp),
body=Expr, orelse=Expr)

# this if for a.b
Attribute(value=Identifier, attr=Identifier)

FunctionReturn(value = Expr)
```

Figure 12.10: Semantic Model for semirings.

12.2.3 Implementation of the DSEL

We follow a similar strategy to the filtering DSEL to implement dynamic translation and compilation for `UnaryFunction/BinaryFunction` objects. Users create a new class that inherits from our special `PcbFunction` class and implement a `__call__` method; this is the standard method used by Python for callable objects. Then, on instantiation, the code contained in this method is translated and compiled, using the intermediate representation to ensure the defined method is

```
class Select2nd(PcbFunction):
  def __call__(self, x, y):
    return y

class Min(PcbBinaryFunction):
  def __call__(self, x, y):
    if (x<y)
      return x
    else
      return y

semiring = kdt.sr(Min(), Select2nd())
```

Figure 12.11: Example of using our DSEL to create a semiring for Breadth-First Search.

translatable and correct. Currently, users must pass in the input and output types during instantiation because no attempt to infer types is made, though the DSEL could be extended to detect run-time concrete types before specialization.

Note that the Semantic Model here restricts operations only to ensure that the translation is correct. In other words, we do not attempt to determine whether the operations are in fact commutative or otherwise will result in a correct semiring.

The translated code is compiled into a dynamically link library that contains a method which instantiates a C++ instance of the appropriate C++ KDT class. In order to interface with KDT properly, KDT's underlying function classes are modified so that customized functions can be inserted into the instances; these are just function pointers that are checked when the instance is called. If the function pointers are non-null, the custom function generated by our DSEL is called; otherwise, the usual KDT method is used. An alternative that was considered was to subclass KDT's C++ `UnaryFunction/BinaryFunction`, but this was rejected because it would force all type specialization to occur at code generation time. The approach we chose is flexible enough to handle further specializations during execution, if needed.

If at any point in the translation/compilation there is a failure, then the execution path resumes using the usual KDT method of wrapping pure Python functions. Since our DSEL uses the same interface as Python for callable objects, we simply pass the user-defined Python instance to the existing KDT infrastructure after printing a warning that translation/compilation failed.

12.2.4 Experimental Results

Lines of code for the semiring operation DSEL is shown in Table 12.6. Because much of the infrastructure for processing the Semantic Model is shared with the filter DSEL, the size of the compiler is very small. This shows the productivity aspect of Asp: it enables writing very small DSEL compilers that can effectively translate domain-specific Python code to low-level C++ .

To test the effectiveness of our DSEL approach, we first measure the performance difference in the underlying sparse matrix-vector multiplication using the semiring operations used in Breadth First Search for our Twitter data structure. We compare the performance of three different ways to write semiring operations in KDT: defining all operations in Python, using a hand-written C++ semiring (which is the default mechanism used for BFS in KDT), and using our DSEL to

	KDT Pure	KDT+C++	KDT+SEJITS
SpMV Semiring	Python	C++	C++
Prune Frontier (Op)	Python	Python	C++
Prune Frontier (check)	Python	Python	C++
Parent Update	C++	C++	C++

Table 12.5: For the three BFS implementations in our experimental study, this table shows which operations occur in which language. KDT+C++ is the default implementation used in the current KDT release.

	Shared	Python	C++
Semiring Op DSEL	82	104	38

Table 12.6: Lines of code for the semiring operations DSEL. Much of the translation infrastructure is the shared with the filter DSEL, making the implementation quite small.

define semiring operations that are then translated into C++ .

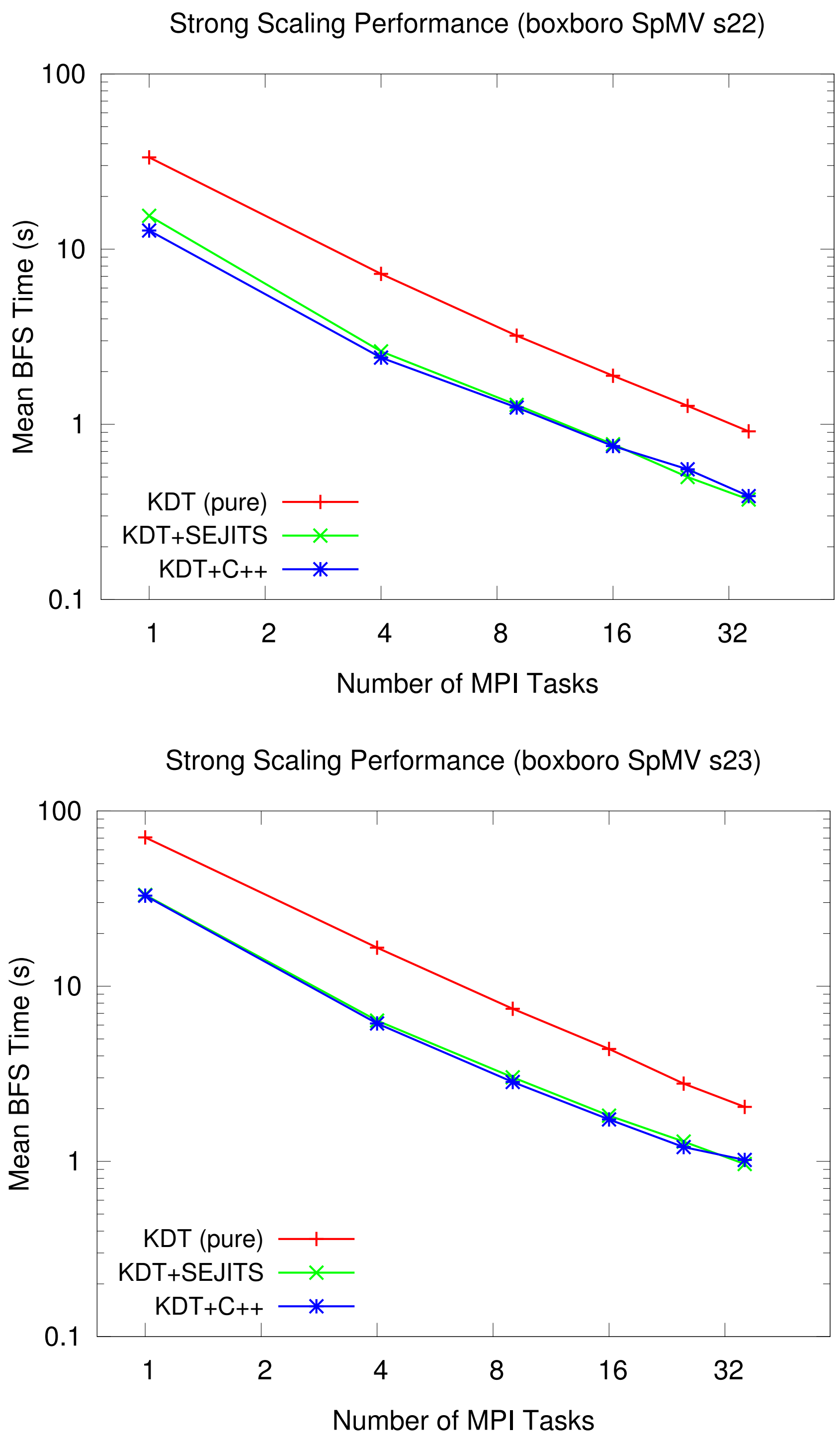
We then compare performance for the full BFS using the three approaches. The algorithm we use is the KDT implementation of Breadth First Search, which we customize to use the appropriate semiring implementation, and run on synthetic Twitter datasets as before using the same parameters for the R-MAT generator.

Finally, we also demonstrate the performance benefits by changing one of KDT's built-in graph algorithms to use our DSEL. The connected components algorithm uses a graph traversal similar to BFS, and finds the set of connected components– that is, each subgraph that is connected but does not connect to other vertices in the graph. Unlike the BFS, however, it operates only on non-semantic edges. Furthermore, it repeats traversals until all vertices are part of a connected component.

Breadth-First Search Results

Note that the KDT implementation of BFS uses several instances of Python functions wrapped in C++ in order to perform the various operations. The semiring used for SpMV uses `(min, select2nd)` built into the Combinatorial BLAS for the add and multiply, and the input and output types are different for the two operations. In addition, the vector-vector operation that removes discovered vertices from the frontier at each step uses a `BinaryFunction` as well as a binary predicate, for which we leverage the DSEL in the previous section to translate into C++ . Finally, the update to the parents vector uses a C++ function for indexing. Table 12.5 shows which operations occur in which language for the three BFS implementations.

In Figure 12.12, we compare the performance of sparse matrix vector multiply with our three implementations of the `(min, select2nd)` semiring. The SpMV calls the underlying semiring for each operation. Performance results show excellent scaling for all three implementations, but in absolute terms the C++ implementation and the DSEL-generated implementation outperform the pure Python version by $2.45\times$ at scale 22 and $2.12\times$ at scale 23. There are some performance differences between the C++ semiring and the DSEL version due to overheads of calling into a dynamic-library function as well as overheads incurred by the check for whether a generated function exists; this results in up to an 18% slowdown at low numbers of processors. Nevertheless,


Strong Scaling Performance (boxboro SpMV s22)

Strong Scaling Performance (boxboro SpMV s23)

Figure 12.12: Performance of sparse matrix vector multiply (SpMV) using three different implementations of the `(min, select2nd)` semiring (strong scaling). These runs use a generated R-MAT matrix of scale 22 (top) and scale 23 (bottom).

	KDT Pure	KDT+SEJITS
SpMV Semiring	Python	C++
Prune Frontier (Op)	Python	C++
Prune Frontier (check)	Python	C++
Parent Update	C++	C++

Table 12.7: For the two connected components algorithms in our experimental results, this table shows the languages used for each operation. Note that since the semiring is different from the BFS semiring, it is not by default defined in C++ .

at the highest concurrencies, the performance is essentially the same. Thus, our DSEL for semiring operations is able to eliminate all of the overhead caused by defining the operations in Python, for the SpMV.

We expect this performance improvement to be less than that of the filter case, because in the filter case, the actual edge payload (data) must be examined by the filter, while in this case, the data carried on the edge is not touched. Thus, the overall data movement is less, giving less opportunity to speed up the operations. The resulting speedup over Python for the semiring operation case, however, is still impressive.

Figure 12.13 shows the performance results for running our Breadth First Search experiment on Boxboro using an RMAT matrices of scale 22 and 23. BFS using our DSEL is $2.3\times$ faster than the pure Python version and even marginally outperforms the existing BFS (which uses semiring operations defined in C++). With our DSEL, we are able to eliminate the overhead of defining the semiring operations in Python, allowing users to write new algorithms completely at the Python level without sacrificing performance.

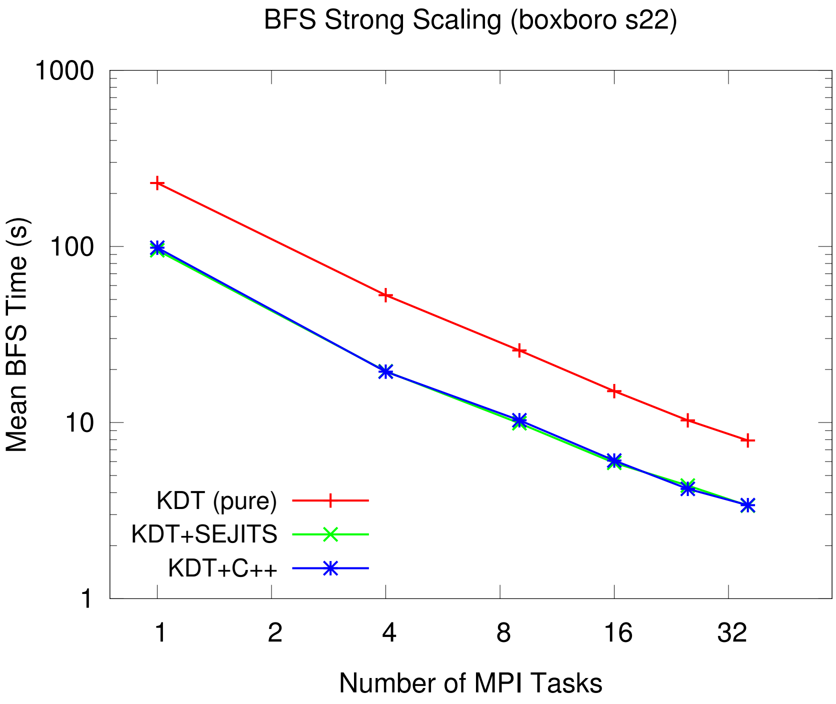
Connected Components Results

The default version of connected components included in KDT uses a slightly different semiring than the one used for BFS, and thus, the semiring is defined in Python instead of C++ . Table 12.7 shows which operations occur in which language for the two connected components implementations we compare.

Figure 12.14 shows strong scaling results on Boxboro for the connected components implementations, with time for copying and symmetrizing the matrices elided. On average, our DSEL implementation is about $2\times$ faster, and for some cases, up to $2.6\times$ faster. These results show that our work is applicable to many of the algorithms in KDT, and by simply changing each algorithm to use our DSEL, we can obtain large performance improvements with little effort.

12.3 Future Work

Future work must address some limitations of the current infrastructure. Our implementations of the DSELs do not automatically type-specialize for the different types present in KDT. Such functionality is clearly desirable because it would allow the same Python code to work for all possible types (instead of forcing users to specify types at filter/semiring instantiation time, and then to use the appropriate instance), but would require integrating some kind of type inference (such



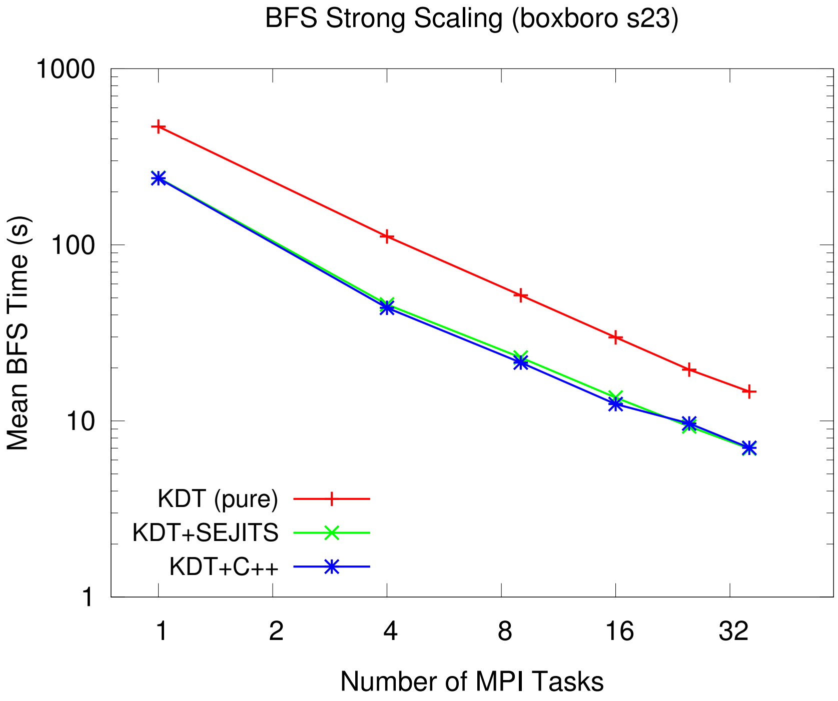
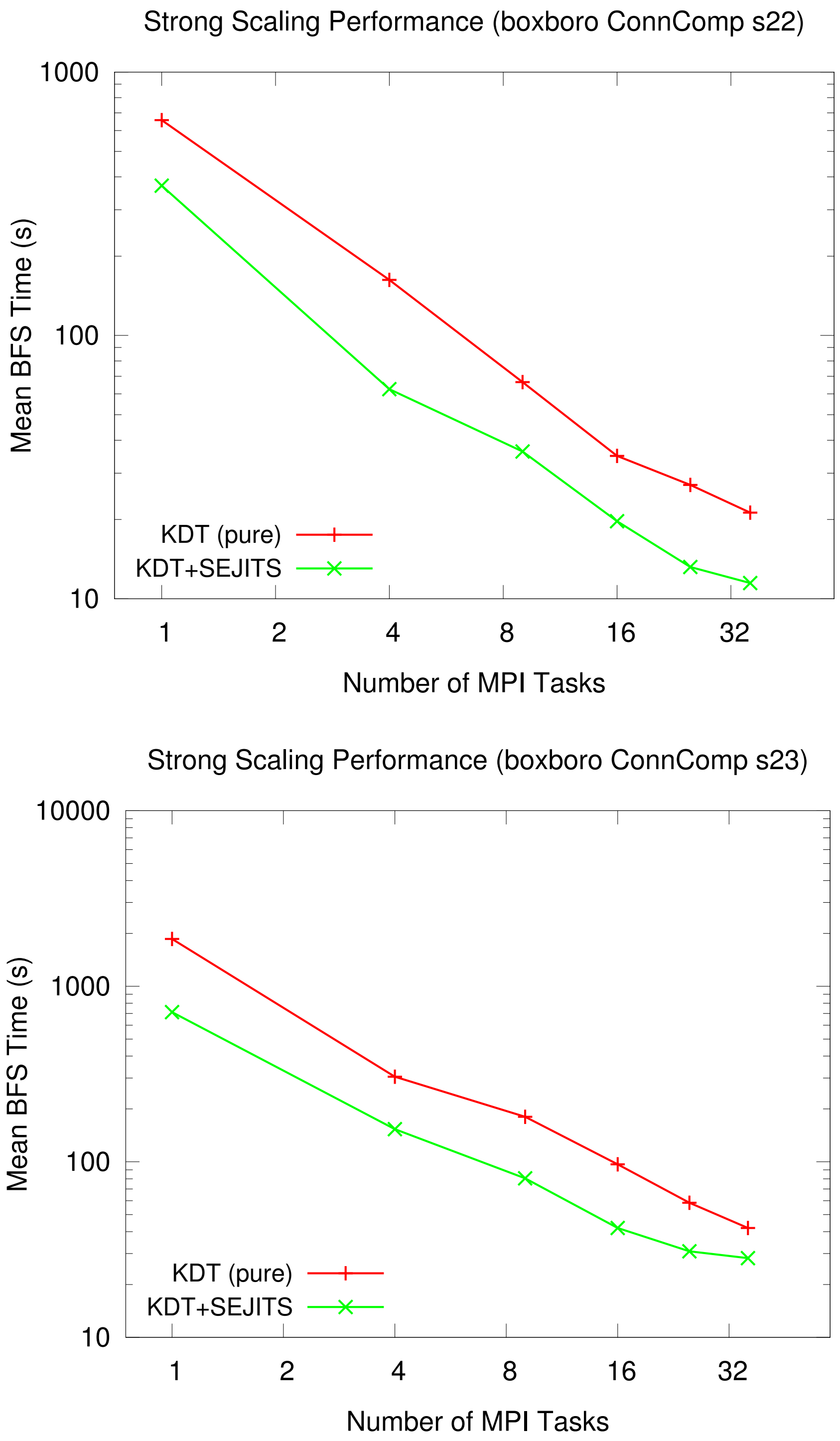


Figure 12.13: BFS strong scaling performance on Boxboro using three different semirings: KDT with pure Python semirings, KDT with our DSEL for semiring operations, and KDT using a semiring hand-coded in C++. The graphs are generated RMAT matrices using scale 22 (top) and scale 23 (bottom).


Strong Scaling Performance (boxboro ConnComp s22)

Strong Scaling Performance (boxboro ConnComp s23)

Figure 12.14: Strong scaling performance for the two implementations of the connected components algorithm, one in pure Python and the other using our DSEL. Results are shown on Boxboro with R-MAT matrices of scale 22 (top) and scale 23 (bottom).

as the tracing inference included in Asp) into the DSEL compilers. In addition, the DSELs could be extended to allow users to call a limited set of C++ native functions, such as the C++ standard library's templated `max()`, which would leverage existing well-optimized C++ code in defining semiring operations.

The current DSELs do not support auto-tuning, mostly due to the fact that they interject themselves into C++ in areas where auto-tuning would not be beneficial. In future work, changing the interjection point to the actual matrix-matrix, matrix-vector, and vector-vector operations would expose areas where auto-tuning is beneficial. Since SpMV auto-tuning already has a rich history [115], the DSELs could auto-tune a specific SpMV implementation that is specialized for data structure, specific semiring operations, and filtering. SpMV auto-tuning in libraries such as OSKI do not optimize computations that do not use the usual addition and multiplication in their semirings. In addition to generating code for serial SpMV, the DSELs could, with appropriate modifications to the Combinatorial BLAS, generate hybrid shared-memory/MPI code for the SpMV, again increasing the scope of potential optimizations.

Finally, one important avenue of future work is to allow users to define custom vertex and edge types in Python and have them automatically translated to C++ for use with the underlying Combinatorial BLAS framework used by KDT. Currently, these base types must be defined in C++ as part of the KDT infrastructure. Asp does not currently contain mechanisms to support such type declarations, but this is a direction for future work in Asp that can then be utilized in KDT.

12.4 Summary

In this chapter, we demonstrated how the SEJITS approach uses translation of domain-specific embedded languages to mitigate performance slowdowns due to programming in a high-level language, while interfacing with an existing Python and C++ package. We did this by defining two DSELs, one for filtering semantic graphs, and one for defining semiring operations in Python. The first enables users to perform graph algorithms on a subset of the graph, without incurring huge performance penalties due to upcalls into Python. The semiring DSEL enables writing new algorithms in KDT at the Python level without sacrificing performance. Together, these advance the ability of users and algorithm designers to write high performance graph analyses with high productivity.

Chapter 13

Other Case Studies: Implemented Domain-Specific Embedded Languages and Auto-tuned Libraries Using the Asp Framework

Along with the more in-depth case studies already presented, this chapter summarizes three other packages that use the Asp framework. In the previous case studies, we implemented domain-specific embedded languages using Asp, but even if only the code generation and auto-tuning facilities are used, efficiency programmers can build auto-tuned libraries for high-level languages that leverage Asp's capabilities. This allows efficiency programmers to use high-level code to implement their libraries, and makes the resulting libraries accessible for productivity programmers who wish to use the high-level embedding language.

Each of the three case studies in this chapter was implemented by other researchers, but all use the Asp framework, demonstrating its usefulness and generality across computational domains and across a variety of parallel platforms.

In Section 13.1 we outline an auto-tuned parallel library for the matrix powers kernel, a building block of high performance communication-avoiding algorithms. Section 13.2 describes an auto-tuned library for Gaussian mixture modeling that targets both multicore CPUs and GPUs. These two libraries use Asp's auto-tuning and code generation support to deliver high performance libraries for Python. For a final example, Section 13.3 describes a domain-specific embedded language for the Bag of Little Bootstraps statistical machine learning algorithm, targeting cloud computing as well as local parallel computation. Section 13.4 summarizes.

13.1 Auto-tuned Matrix Powers for Python

Communication-avoiding algorithms [7] strive to increase performance of computations by modifying them to reach the theoretical minimum communication bounds for the amount of data movement required, often at the cost of increased computation. Given the historical trends of memory performance relative to processor performance, minimizing data movement is increasingly the best mechanism for optimizing algorithms on future architectures.

Optimization	Type
Thread blocking	Re-ordering
Explicit cache blocking	Re-ordering
Tiling	Size reduction
Symmetric representation	Size reduction
Implicit cache blocking	Re-ordering
Index array compression	Size reduction

Table 13.1: Summary of optimizations for matrix powers. Some optimizations re-order the computation to improve memory traffic usage; others reduce the amount of memory traffic by reducing the storage size of the matrix.

One area of computation to which communication-avoiding optimizations have been applied is solvers for linear systems. Such algorithms, given a matrix A and vector b, solve the equation $Ax = b$ for x. When the matrix A is large and sparse, direct solvers (such as Cholesky factorization) can be impractical, so a number of iterative methods have been developed for such cases. One class of these algorithms is Krylov subspace methods (KSMs), which can also be used to find eigenvalues and eigenvectors for a matrix (that is, for a given matrix A, the λ and x such that $Ax = \lambda x$).

Within KSMs, the major operation is sparse matrix vector multiply (SpMV) which occurs in every iteration of the algorithm. For communication-avoiding KSMs [51, 81], the SpMV is replaced by a kernel that performs k SpMVs at once, called the *matrix powers kernel* or the $A^k x$ kernel. This refactoring enables implementations to divide the matrix into cache- or local memory-sized blocks and reuse the entries of each block k times, substantially reducing overall memory traffic but incurring redundant computation. In addition to changing the SpMV portion, the overall solver algorithms usually need to be refactored to change some of the other operations to preserve correctness and to take advantage of resulting opportunities for optimization.

Because there is a tradeoff in the parallel implementation between reducing memory traffic and increasing redundant computation, to ensure the best possible performance, the matrix powers kernel requires a large amount of tuning. In this section, we outline an auto-tuned matrix powers kernel for Python that allows users to build communication-avoiding KSMs that obtain high performance. To ensure interoperability with existing Python code, the kernel interfaces with NumPy, the widely-used numerical library for Python. For the user, the interface is kept as simple as possible: they call the auto-tuner with a particular matrix structure, and, after the auto-tuner is finished trying variants, it returns an object representing the optimization plan that yields the best performance. This object can then be used to call the kernel for subsequent runs (and can be preserved between application invocations).

13.1.1 Implementation Strategy

The auto-tuner described here was implemented by Jeffrey Morlan for his master's thesis [84] at U.C. Berkeley, using the Asp framework described in this work.

The auto-tuner works by implementing optimizations that reduce memory traffic as well as optimizations that re-order the computation in ways that improve parallelization or better use the cache. These optimizations are summarized in Table 13.1 and described in more detail in [84].

Code generation is handled by a set of parameterized Asp templates, C++ code with Python

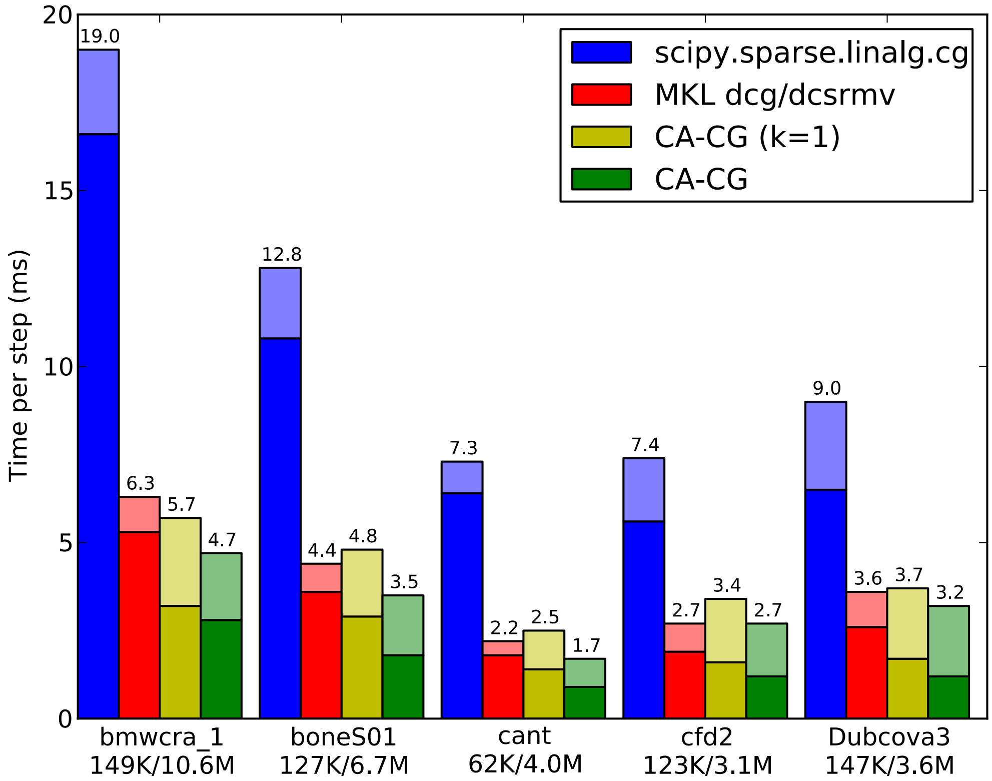


Figure 13.1: Performance of CG versions on an Intel Xeon X5550 (8 cores, 2.67 GHz).

control code interspersed within, controlling how the code generation proceeds. Parameters to each of the optimizations are controlled by the tuning system, and are set depending on properties of the input matrix as well as the machine architecture.

The auto-tuner generates a set of candidate parameters for the optimizations, and then generates code for each combination. Each of these candidate implementations is compiled through the Asp infrastructure into a dynamic library, and run to determine empirical performance. Once all candidates have been exhausted, the optimization plan representing the fastest is returned for subsequent use by the application. When the kernel is called with this plan, only the fastest is compiled (if needed) and run. This exhaustive search is clearly overkill for the auto-tuner; future work can limit the space searched by the tuning system.

13.1.2 Performance Results

The test solver application for this auto-tuner is the communication-avoiding conjugate gradient (CA-CG) algorithm [51], which is a Krylov solver that operates on symmetric positive definite systems (that is, the matrix A representing the system is symmetric and, for all non-zero real vectors z, $z^T Az > 0$). The performance of the CA-CG implementation is compared against the parallel CG implementation in SciPy, a scientific computing library for Python, as well as with the parallel CG from Intel's MKL library [54].

Performance results on an 8-core Intel Xeon X5550 are shown in Figure 13.1. The dark portion of each bar shows time spent in the matrix powers kernel. In every case, the communication-avoiding version outperforms both the SciPy version and Intel's highly-optimized MKL implementation. In these charts, however, the tuning time is elided, and the time is on the order of a few thousand calls of the kernel. This time can be improved greatly, however, with better heuristics for determining which portion of the space to explore.

The tuner uses two major features of Asp: the ability to express code snippets with inline Python control code and support for auto-tuning. Overall, the matrix powers auto-tuner demonstrates how Asp and SEJITS can be used to deliver auto-tuned libraries that outperform highly hand-optimized linear algebra routines, even if they do not use the full power of the approach.

13.2 Gaussian Mixture Modeling for CPUs and GPUs

Gaussian mixture models (GMMs) are a type of probabilistic density model made up of Gaussian component functions. Such models are widely used in a variety of domains, including speech-related recognition, financial modeling, and handwriting recognition. To find the component functions that fit an observed data set, the standard method used is the Expectation-Maximization (EM) [82] algorithm, a highly compute-intensive iterative method.

The EM algorithm proceeds by alternating between the Expectation (E) step and Maximization (M) step. During the E step, given the current parameter estimates, the algorithm computes the expectation of the log-likelihood of the observations under study. During the M step, parameters are found that maximize the log-likelihood found during the E step. The iterations proceed until the model parameters satisfy the given convergence criteria.

In this section, we outline the implementation of an auto-tuned version of the EM algorithm for calculating GMM parameters that allows users to call a single Python function which generates and compiles parallel code on multicore CPUs and GPUs and executes the EM algorithm, returning the resulting parameters to the user. Because of its high computation requirements, the EM algorithm highly benefits from parallelization; however, the strategy for the best parallelization differs given characteristics of the observations, characteristics of the model being created, and the computational capabilities of the machine. The auto-tuned library described in this section was built by Henry Cook and Ekaterina Gonina at U.C. Berkeley using the Asp framework and is available for download. [1]

13.2.1 Implementation Strategy

The auto-tuned GMM implementation uses templatized code snippets interjected with Python control flow to decide how code generation occurs. Based on input parameters, different implementations are used; for each possible parallelization strategy (the combination of which platform to target, plus on which axes the parallelism should occur) a different code generator is utilized. Utility functions and those required to interface between Python and the generated code are statically included in the implementation. These include the necessary functions for loading data onto the GPU and back when generating GPU-targeted code.

For backend hardware, the GMM tuner targets CUDA-compatible Nvidia GPUs and x86 multicore processors that can use Intel's Cilk Plus runtime. Generally speaking, current high-end GPUs have greater computational capability than can be obtained with current multicore processors;

[1] Source code available at `http://github.com/hcook/gmm`.

however, the required data movement to and from GPU memory can eliminate any potential performance gains from using the GPU as a coprocessor.

The different implementations on each backend differ primarily in how the parallelization occurs. More specifically, the different versions choose a different nesting of the parallel loops, and generate code that is correct given the chosen nesting. In addition, further variants use blocking to improve performance of the parallelization.

Currently, the auto-tuner tries all correct variants for a particular set of problem parameters; this is done similarly to the structured grid DSEL in Chapter 10 so that each time the function is called with the same set of parameters, a different variant is chosen until all variants have run. Subsequent calls with the same parameters will always use the fastest, even across program invocations.

13.2.2 Performance Results

The application for testing performance results is a speaker diarization application. The goal of this program is to determine which speaker says what in a recorded meeting, and proceeds by using GMMs within a segmentation, training, and agglomeration loop; details of the algorithm used are in [4]. For the purposes of evaluation, diarization applications are usually judged using two metrics: time for completion and error rate. In this evaluation, we elide discussion of error rates, but note that there generally is a tradeoff between the two evaluation criteria: the longer you run the diarization, the more accurate the results will be, until some limit is reached.

Gonina et al [43] found that their implementation of the overall diarization algorithm in Python outperformed the original C++ with pthreads code by a factor of $3\times$ on multicore CPUs, and up to $6\times$ on an Nvidia GTX480 GPU. The lines of code for the application also shrunk dramatically, due to the expressiveness of Python as well as the ability to leverage high-level libraries.

A comparison of raw performance for the GMM portion of the computation is shown in Figure 13.2 on a dual-socket Intel X5680 3.33 GHz machine and on an Nvidia GTX480 GPU. The raw performance for the CPU is increased by almost $10\times$ in some cases, while the tuned GPU implementation outperforms a hand-written low-level CUDA implementation for some of the datasets. For the $M = 2$ dataset, the use of a CUDA-based version is unnecessary on this hardware, as the multicore performance is very close to the tuned CUDA performance.

In all, the performance results show that the combination of auto-tuning and run-time code generation can yield excellent performance for computationally-intensive kernels, and can be made portable across execution platforms. To users, it appears that they just call a single black-box function, but that function can choose where to run based on available hardware and problem parameters.

13.3 A DSEL for the Bag of Little Bootstraps Algorithm

Bootstrapping is a statistical method that attempts to measure the quality of a statistical estimator when only an approximate distribution is available, and is based on using random sampling with replacement to simulate sampling from a larger population than is available. The Bag of Little Bootstraps (BLB) algorithm [64] is a variant of general bootstrapping that exposes more parallelism and can potentially run in much less time.

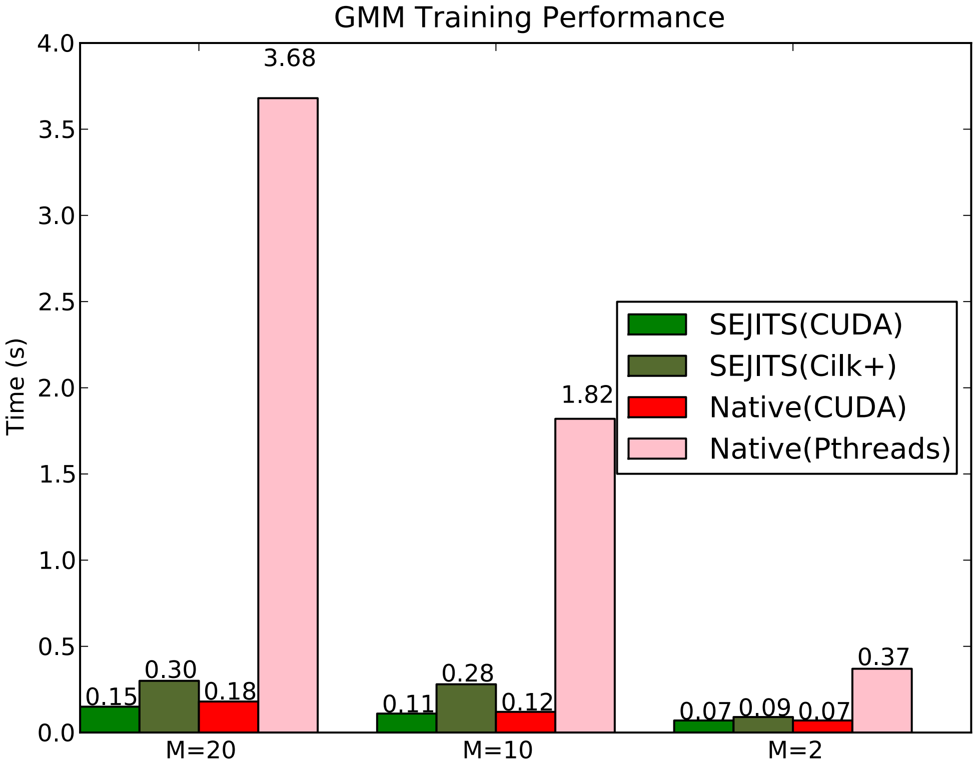


Figure 13.2: Raw Gaussian mixture model training performance on an Intel X5680 3.33 GHz CPU and Nvidia GTX480 GPU. The versions using the SEJITS approach outperform the native (original) implementations due to using auto-tuning.

Similar to the structured grid computations described in Chapter 10, this algorithm is not well-suited for packaging in an auto-tuned library due to the use of application-dependent *estimator* functions and *reducer* functions. The parallelism in BLB can be implemented on shared memory as independent computations distributed over the processors in the system, all reading from the same data store, while in the cloud, the data must be partitioned among the available machines to ensure acceptable performance with large data sizes.

This section describes a high-performance DSEL for BLB in Python [95], using the Asp framework, implemented by Peter Bersinger, David Howard, Aakash Prasad, and Richard Xia at U.C. Berkeley. The DSEL can use either a multicore CPU via Cilk Plus for execution, or, if the data size requires, execute in the cloud using the Spark [125] infrastructure. Note that the latter requires generating Scala code, while the Cilk Plus implementation uses C++ with parallel extensions. Thus, this DSEL is an example where user-supplied code is translated into different languages based on where the execution occurs, and is an example of cloud computing support in Asp-based DSELs.

13.3.1 Implementation

Users specify the estimators and reducers for their problem by subclassing a specific parent class and writing two functions, one each for the estimator and reducer, in a subset of Python, similar to the interfaces exposed by the structured grid DSEL in Chapter 10. Within each of these functions, a large subset of mathematical operations supported by pure Python can be used, including operations such as mean and exponentiation, since these are commonly used in estimators and reducers.

The user-supplied function is parsed into a Python AST by the Asp framework, and this AST is then translated to a Semantic Model that describes supported computations. The Semantic Model is used to generate either Scala code for use with Spark in the cloud, or Cilk Plus code for use on a shared memory machine. Both backends support the same kinds of BLB problems. In addition to generating the estimator and reducer functions, the DSEL framework also outputs code to interface between the parallel code and Python. In the case of Cilk Plus, these are simple conversions between NumPy datatypes and C++ datatypes when necessary, but in the case of the Scala backend, the DSEL uses Asp's Scala support to translate datatypes using the Apache Avro [6] data serialization library. Along with code to translate between Scala and Python, the DSEL also executes the call to the remote Spark instance and processes the results. To end-users, the execution appears the same whether pure Python, Cilk Plus, or Spark is used.

13.3.2 Performance Results

To measure scaling for the BLB DSEL, the DSEL authors perform model verification of an SVM classifier on a randomly-selected subset of the Enron email corpus, consisting on over 126,000 feature vectors and tags, with each feature vector composed of approximately 96,000 features. 10% of the subset was used for training the classifier, and the remaining 90% was used as a test set.

On a system with 4 Intel X5760 processors, the algorithm was able to scale 31.6× serial performance on 32 cores, with almost no accuracy loss. This almost-ideal scaling is due to the BLB algorithm's excellent parallel structure, which the DSEL is able to fully utilize when generating code.

The BLB results show that the same DSEL can be used for generating high performance code for both shared memory machines and for remote execution on the cloud, and scale almost ideally if the algorithm is amenable to it, bringing high performance in parallel to domain scientists using statistical methods.

13.4 Summary

This chapter described three projects using Asp that have been implemented by others who are not primary authors of Asp. Collectively, the three demonstrate that Asp and the SEJITS approach can deliver high performance, excellent parallel scaling, and domain-specific auto-tuning to users of high level languages. Furthermore, since all three of these were developed by people outside the core Asp developers, they show that the Asp infrastructure is usable for efficiency experts for developing auto-tuned libraries and DSELs.

Chapter 14

Insights, Future Directions, and Conclusions

In this chapter, we step back and view the results of our case studies from a high level, synthesizing lessons we can take from them as well as exploring some future directions for research. The work here has shown the potential of our approach in bridging the Performance-Productivity Gap, and the further directions outlined in this chapter will help demonstrate that potential. We finish by summarizing the contributions of this thesis.

Section 14.1 outlines some observations from the case studies and high level conclusions we can draw from them. In Section 14.2 we discuss how to extend the number of DSELs using our methodology, and Section 14.3 discusses one of the major remaining obstacles before out desired pattern-based programming approach can be implemented. In Section 14.4 we outline some directions for Asp development. Section 14.5 concludes the thesis, summarizing the many contributions.

14.1 Insights from Case Studies

We have presented six case studies, each with varying goals and varying scope in terms of the breadth of the DSEL or auto-tuner. The different case studies operate over different segments of their respective domains, from larger scopes such the structured grid DSEL to libraries that implement a single operation, like the matrix powers library. This set of case studies covers a wide swath of potential use cases for the SEJITS approach.

Three of our case studies strive to obtain the highest possible percentage of peak performance. For the structured grid DSEL, we are able to write a simple compiler that, using domain-specific optimizations and auto-tuning, can obtain over 93% of peak performance. The matrix powers kernel outperforms currently-available Python libraries and even beats optimized vendor-provided libraries across architectures. The Gaussian mixture modeling library outperforms existing versions of the algorithm and can selectively target either an available GPU or run on a multicore machine. With these three case studies, we demonstrate that the SEJITS approach can be used to deliver auto-tuning both for traditional libraries and for motifs for which writing libraries is difficult. The packages built with SEJITS are able to obtain high portions of absolute peak performance.

The goal in three other case studies is to eliminate the overhead of using higher-level functions in frameworks where both performance is important and user-defined functions are essential. The KDT DSELs interface with an existing multi-layered framework with distributed parallelism, yet still eliminate almost all of the overhead of using Python. The Bag of Little Bootstraps implementation desires to use real parallelism within a node and interface with the existing Spark infrastructure on the cloud, even though the latter is written in Scala, a language that is not easily interoperable with Python. In all three cases, the implementations greatly reduce the overhead of using a high level language and are able to interface with existing frameworks that control parallelism.

In our approach, we try to eliminate analysis as much as possible by implementing declarative languages that are expressed in an imperative syntax. Using the declarative approach, correctness can be determined during the transformation from Python syntax to our intermediate representation; in most cases, no further analysis is necessary. Further, by making our DSELs declarative, we concentrate on enabling users to supply compilers with what the computation should be, not how to perform it. Our DSEL compilers use knowledge of the domain combined with knowledge of the backend execution units to automatically determine how to efficiently compute the user-supplied code.

The use of Asp, our infrastructure for productively building DSELs in Python, also yields some insight. First, all of the case studies are very small in terms of lines of code required for logic; the largest ones use templating and most of the code involves low-level templates. Indeed, the productivity of writing code in high level languages is demonstrated by how small the DSEL compilers are. Furthermore, Asp eliminates many operations that DSEL designers otherwise would need to care about, such as auto-tuning and compiling. It also makes other things easier: defining the intermediate representation, interfacing with python, and caching to eliminate the need for compilation when the code has already been used once.

Overall, the six case studies using Asp span a large number of use cases and provide evidence that our approach to DSELs can result in productive, fast, parallel code for users, with a low threshold effort for DSEL implementers.

14.2 Future Directions: Building an Ecosystem of DSELs

In our future vision of pattern-based programming, productivity programmers select which high-level patterns their application is composed of, and use frameworks, libraries, and DSELs that implement subsets of each pattern in order to build their application. For this vision to succeed, we must have enough DSELs that large and complicated applications can be architected from them. A good test of this approach will be when enough DSELs exist that a full application can be constructed from multiple DSELs. One future direction, then, is to build up a suite of DSELs and libraries using our approach for use by programmers.

Even for existing DSELs, extensions that increase their scope or implement new optimizations are necessary. For example, the structured grid DSEL could be made more functional with some language extensions, or could optimize code for the small grids present in Adaptive Mesh Refinement, which require a very different set of transformations and optimizations than the class of structured grid optimizations currently implemented.

For this large-enough suite of DSEL compilers to exist, a community of developers is necessary. For languages such as Python and Ruby, programming communities continue to write innovative new software packages that greatly extend the usefulness of each language. If we were able to do

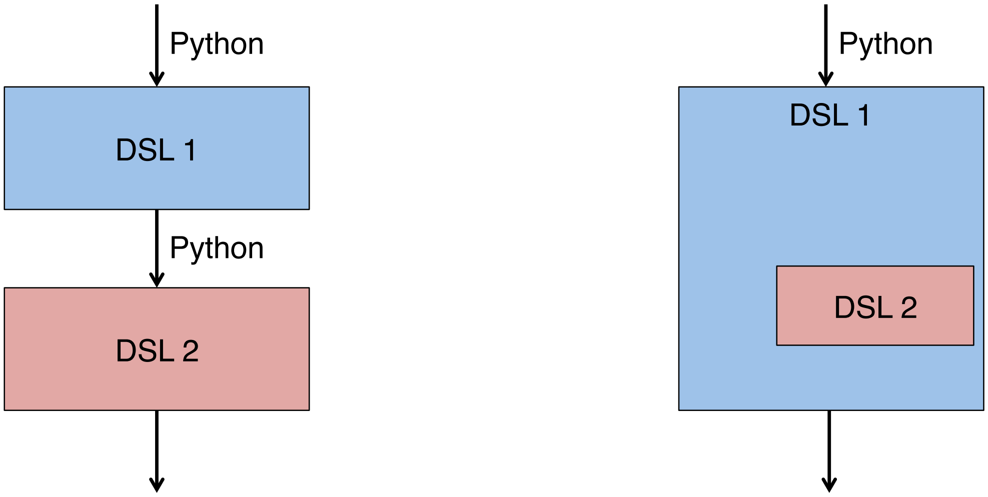


Figure 14.1: Two kinds of composition for DSELs. Left: a single Python program uses two different DSELs, passing the output of one to the other, perhaps with some code to transform one's output to the other's input. Right: a single Python program uses two different DSELs, with one DSEL using the other as a subroutine.

the same around pattern-specific DSELs, the momentum would be useful for both extending exiting DSEL compilers and for building new ones as they become necessary, essentially "crowdsourcing" the continued expansion of the DSEL suite . A central repository such as the Python Package Index (PyPI) [99] or RubyGems [34] would help publicize and build the usefulness of our DSELs.

14.3 Future Directions: Composing DSELs

Another necessary piece of functionality for our future vision of programming is the need to compose pattern-based DSELs and libraries. Note that this need not be general all-to-all composition, since compositions between certain patterns doesn't make sense. But for combinations where composition does make sense, we see two different kinds of composition: sequential and subroutine. Figure 14.1 shows the two kinds of composition. In sequential composition, DSELs are used one after another, with intermediate code possibly coordinating between the two. In contrast, the subroutine composition case occurs when one DSEL is used as a subroutine in another.

It is rather simple to build infrastructure for sequential composition, which mostly already works. However, data structure interoperability is not automatic and requires either that both DSELs use the same underlying data representation or that the user explicitly transform the data . Future extensions to the infrastructure could make interoperability easier if some mechanism that enables separate DSELs to agree on JIT transformations between representations as necessary, or to agree on tuning decisions that impact the performance of each. Such agreement on tuning decisions between DSELs is necessary for good performance, as shown in previous work on co-tuning (auto-tuning when two different kernels are involved, each with their own tuning decisions that impact the other) [80].

Subroutine composition, however, is more difficult. The locus of control is now the "outside" DSEL, and the "inside" DSEL must have some functionality for allowing tuning decisions to be made outside of it. In other words, we must build DSELs that have three modes: one where the

DSEL controls everything, one as where it is the outside DSEL controlling decisions by subroutines, and one as the inside DSEL. Even with this strategy, many potential issues remain, such as how to generate interoperable code and how exactly tuning decisions will work in this context. This remains a major area of future work necessary to enable more powerful composition. One approach is to begin by building related DSELs for a computational domain that can be composed with one another; by limiting the interoperable DSELs, the composition problem becomes more tractable. Already, preliminary work building a set of DSELs for multimedia content analysis applications is starting to explore issues of composition in more detail [44].

14.4 Future Directions for Asp

Asp has shown successfully that infrastructure plus a high level language can enable productively building DSELs that are small and result in portable performance. However, much more can be done to make building DSELs using the SEJITS approach easier and to enable DSEL designers to extend the functionality they can include in their packages and frameworks.

14.4.1 Data Structure Definitions

One common task that Asp does not support but would allow further functionality is defining user-defined data structures in Python that are automatically translated into C++ structures that are accessible from Python, but can be used in the code generation toolchain. For example, recall that KDT forces users to write data structures in C++ if they require custom ones, with a limit of only two types. Asp support for such data structure definitions would be extremely useful. Existing Python libraries can be leveraged, such as ctypes, which is part of the standard library in Python and allows users to interface with C data structures when wrapping external C libraries. Such functionality could be integrated with Asp, perhaps by using a thin DSL that generates ctypes internally.

14.4.2 Improvements in Code Generation

One of the current limitations in Asp is the lack of unification of templates and backend ASTs. Although templates can include ASTs, we do not allow templated code to be transformed or manipulated using the AST transformation framework, due to not including a C++ (or other backend) parser. For this functionality, it may be possible to interface with the Rose transformation framework [33] or to utilize LLVM; as a consequence, we may explore changing the representation for C++ ASTs to one utilized by these packages.

In addition, one promising way to generate code is to use the Sketch [103] system. In Sketch, the compiler takes as input a simple imperative specification written in Java-like syntax, combined with an optimized skeleton with “holes” such as missing integers or under-specified expressions. The compiler then synthesizes the skeleton into an implementation that provably computes the same answer as the specification, for all inputs. DSELs could be implemented with even less effort if we leverage this technology from within Asp and allow DSEL implementers to use synthesis to generate provably-equivalent optimized code.

14.4.3 Compilation As A Service

With mobile devices becoming more and more important to support, Asp can possibly be used to implement compilation-as-a-service: instead of running a compilation toolchain locally, which may consume more power than is desired, the DSEL infrastructure would send appropriate parameters to a web service which returns a runnable, compiled library that can be used. This allows just-in-time specialization to occur without requiring a complete toolchain be present, which can be advantageous for all classes of machines.

14.4.4 Speeding Up Auto-tuning

The major time-consuming portion of DSELs that use auto-tuning is the search. Although we have implemented incremental auto-tuning that does not require large amounts of up-front testing, completely searching the possible parameter space is not necessarily the most efficient way to settle on the best possible version.

We have seen that different strategies to explore the auto-tuning space can converge on a best version more quickly than randomly running all of them. Strategies such as hill climbing or gradient ascent can obtain good results more quickly, especially when the parameter space behaves in predictable ways. Furthermore, machine learning has shown promise [41] for determining best parameters for auto-tuned code. However, the accuracy and usefulness of machine learning is enhanced when more data is available.

One idea is to build a global database that can be used by DSEL auto-tuners to store and retrieve performance information, indexed by architecture and suitable per-DSEL parameters. Such a database could aggregate information from all DSEL compilers and run machine learning algorithms to guess best tuning parameters for new problem instances or new architectures. Incremental revisions to machine learning models enabled by this approach may bring increasing accuracy to subsequent initial guesses.

14.5 Conclusion

In this thesis, we present a diversity of contributions, including a new methodology for combining DSELs and auto-tuning, software infrastructure for building these DSELs that is being used by others, and several examples of DSELs that obtain excellent performance compared with the state-of-the-art across differing domains, thanks to aggressive domain-specific optimizations encoded in the DSEL compilers. Specifically, the contributions include:

- A framework in Chapter 5 for writing DSEL compilers through a set of abstractions for manipulating and generating code and demonstrate their use in building auto-tuners, both by the original authors of the framework and by outside users.
- We develop a technique for auto-tuning computations that involve higher order functions, i.e., stencils or graphs, that can only be tuned after instantiation with a user-provided operator, in Chapters 10 and 12.
- This technique uses an intermediate representation based on declarative semantics which provides the freedom needed to transform code without the need for difficult analyses.
- We demonstrate, in Chapter 10, a simple imperative language for expressing structured grid computations that is translated using introspection into a declarative intermediate form that allows for a large set of possible implementations. We demonstrate the high level interface and its restrictions, which eliminate the need for complex analysis. The results show performance that obtains over 83% of peak memory bandwidth across different machines for a variety of kernels.
- We show a second case study in Chapter 12 of graph traversal algorithms that uses an existing hand-tuned library (CombBLAS) and solves an important performance problem of optimizing over user-provided operators written in a high-level language. The traditional approach of calling back to the high-level language is prohibitively expensive and the DSELs are able to match or exceed the performance of hand-written low-level operators, even without auto-tuning. In addition, the DSELs enable future optimizations that would not be possible otherwise.
- We demonstrate the effectiveness of our framework as a vehicle for delivering library auto-tuning to high-level languages for computations for which a full DSEL is unnecessary, in Chapter 13, with two examples that obtain excellent performance on CPUs and GPUs. We demonstrate that the same DSEL code can be used to execute on multicore CPUs, GPUs, and the cloud. These examples are implemented by outside users, demonstrating that our infrastructure and approach is usable by other performance experts.

As large-scale machines approach peak performance of an Exaflop per second (10^{18} floating point operations per second), resulting in new complexities in architecture, the contributions of this thesis are becoming even more important. Such large increases in computation power will be unavailable to programmers who are not performance experts. Unless the programmability of such machines is increased using methodologies like those presented in this thesis, important simulation problems such as next-generation climate simulation will not benefit from larger, more complex machines. With this thesis, we demonstrate a viable approach to bridging the Productivity-Performance Gap, regardless of complexities in next-generation computer architectures.

Bibliography

[1] David Abrahams and Ralf W. Grosse-Kunstleve. "Building Hybrid Systems with Boost. Python". In: *C/C++ Users Journal* 21.7 (July 2003). URL: `http://www.osti.gov/energycitations/product.biblio.jsp?osti\_id=815409`.

[2] Al Danial et al. *CLOC– Count Lines of Code*. 2012. URL: `http://cloc.sourceforge.net`.

[3] E. Anderson et al. *LAPACK Users' Guide (third edition)*. Philadelphia: SIAM, 1999. URL: `www.netlib.org/lapack`.

[4] X Anguera, Simon Bozonnet, Nicholas W D Evans, Corinne Fredouille, G Friedland, and O Vinyals. "Speaker Diarization: A Review of Recent Research". In: *IEEE Transactions On Acoustics Speech and Language Processing (TASLP), special issue on New Frontiers in Rich Transcription* (2011).

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland, 2009. URL: `http://groups.csail.mit.edu/commit/papers/2009/ansel-pldi09.pdf`.

[6] *Apache Avro*. The Apache Software Foundation, 2011. URL: `http://avro.apache.org/`.

[7] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. "Minimizing Communication in Numerical Linear Algebra". In: *SIAM Journal of Matrix Analysis Applications* 32.3 (2011), pp. 866–901. DOI: `10.1137/090769156`.

[8] Michael Bayer. *Mako : Templates for Python*. 2012. URL: `http://www.makotemplates.org/`.

[9] David Beazley. "Understanding the Python GIL". In: *PyCON Python Conference*. Atlanta, Georgia, 2010.

[10] David M. Beazley. "SWIG: An Easy to Use Tool for Integrating Scripting Languages With C and C++". In: *USENIX Tcl/Tk Workshop*. TCLTK'96. Monterey, California: USENIX Association, 1996, pp. 15–15. URL: `http://dl.acm.org/citation.cfm?id=1267498.1267513`.

[11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. "Cython: The Best of Both Worlds". In: *Computing in Science Engineering* 13.2 (2011), pp. 31 –39. ISSN: 1521-9615. DOI: `10.1109/MCSE.2010.118`.

[12] Fabrice Bellard. *Tiny C Compiler*. URL: `http://tinycc.org`.

[13] Marsha J. Berger and Joseph E. Oliger. *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*. Tech. rep. Stanford, CA, USA, 1983.

[14] Jeff Bilmes, Krste Asanović, Chee-Whye Chin, and Jim Demmel. "Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology". In: *International Conference on Supercomputing*. Vienna, Austria, 1997.

[15] L. S. Blackford et al. *ScaLAPACK Users' Guide*. Philadelphia: SIAM, 1997. URL: `www.netlib.org/scalapack`.

[16] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. "Tracing the Meta-Level: PyPy's Tracing JIT Compiler". In: *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. Genova, Italy: ACM, 2009, pp. 18–25. ISBN: 978-1-60558-541-3. DOI: `10.1145/1565824.1565827`. URL: `http://portal.acm.org/citation.cfm?id=1565827`.

[17] Uday Bondhugula, J. Ramanujam, and et al. "PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System". In: *Programming Language Design and Implementation (PLDI)*. 2008.

[18] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A mUltigrid Tutorial (2nd ed.)* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0-89871-462-1.

[19] Aydin Buluc, Armando Fox, John Gilbert, Shoaib Ashraf Kamil, Adam Lugowski, Leonid Oliker, and Samuel Williams. *High-Performance Analysis of Filtered Semantic Graphs*. Tech. rep. UCB/EECS-2012-61. EECS Department, University of California, Berkeley, 2012. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-61.html`.

[20] Aydın Buluç and John R. Gilbert. *The Combinatorial BLAS: Design, Implementation, and Applications*. Tech. rep. UCSB-CS-2010-18. University of California, Santa Barbara, 2010.

[21] Jacob Burnim, Tayfun Elmas, George C. Necula, and Koushik Sen. "NDSeq: Runtime Checking for Nondeterministic Sequential Specifications of Parallel Correctness". In: *Programming Language Design and Implementation*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 401–414. ISBN: 978-1-4503-0663-8.

[22] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. "SEJITS: Getting Productivity And Performance With Selective Embedded JIT Specialization". In: *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*. Raleigh, NC, 2009.

[23] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. "A Domain-Specific Approach to Heterogeneous Parallelism". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Ed. by Calin Cascaval and Pen-Chung Yew. ACM, 2011, pp. 35–46. ISBN: 978-1-4503-0119-0.

[24] J.C. Chaves et al. "Octave and Python: High-Level Scripting Languages Productivity and Performance Evaluation". In: *HPCMP Users Group Conference, 2006*. 2006, pp. 429–434. DOI: `10.1109/HPCMP-UGC.2006.55`.

[25] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures". In: *IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 676–687. ISBN: 978-0-7695-4385-7. DOI: 10.1109/IPDPS.2011.70. URL: http://dx.doi.org/10.1109/IPDPS.2011.70.

[26] I-Hsin Chung. "Towards Automatic Performance Tuning". AAI3153156. PhD thesis. College Park, MD, USA, 2004. ISBN: 0-496-13683-6.

[27] Peter J. A. Cock et al. "Biopython". In: *Bioinformatics* 25.11 (June 2009), pp. 1422–1423. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp163. URL: http://dx.doi.org/10.1093/bioinformatics/btp163.

[28] Graph 500 Steering Committee. *The Graph 500 List*. 2012. URL: http://graph500.org.

[29] Ral de la Cruz and Mauricio Araya-Polo. "Towards a Multi-Level Cache Performance Model for 3D Stencil Computation". In: *Procedia CS* 4 (2011), pp. 2146–2155. DOI: http://dx.doi.org/10.1016/j.procs.2011.04.235.

[30] Kaushik Datta. "Auto-tuning Stencil Codes for Cache-Based Multicore Platforms". PhD thesis. EECS Department, University of California, Berkeley, 2009.

[31] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine A. Yelick. "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors". In: *SIAM Review* 51.1 (2009), pp. 129–159.

[32] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. "Stencil Computation Optimization and Autotuning on State-of-the-Art Multicore Architectures". In: *Supercomputing*. 2008.

[33] Kei Davis and Daniel J. Quinlan. "ROSE: An Optimizing Transformation System for C++ Array-Class Libraries". In: *ECOOP Workshops*. 1998, pp. 452–453.

[34] Dirk Elmendorf. "RubyGems". In: *Linux J.* 2006.147 (July 2006), pp. 3–. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=1145562.1145565.

[35] Embeddable Common Lisp. *http://ecls.sourceforge.net/*.

[36] Dawson R. Engler and Todd A. Proebsting. "DCG: An Efficient, Retargetable Dynamic Code Generation System". In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, California, United States: ACM, 1994, pp. 263–272. ISBN: 0-89791-660-3.

[37] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture". In: *ACM Transactions on Internet Technologies* 2.2 (May 2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: http://doi.acm.org/10.1145/514183.514185.

[38] Martin Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 2010. ISBN: 0321712943.

[39] Matteo Frigo and Steven G. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005). Special issue on “Program Generation, Optimization, and Platform Adaptation”, pp. 216–231.

[40] Matteo Frigo and Volker Strumpen. “Cache Oblivious Stencil Computations”. In: *International Conference on Supercomputing*. ICS ’05. Cambridge, Massachusetts: ACM, 2005, pp. 361–366. ISBN: 1-59593-167-8.

[41] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. “A Case for Machine Learning to Optimize Multicore Performance”. In: *Workshop on Hot Topics in Parallel Computing (HotPar)*. 2009. URL: `http://www.usenix.org/event/hotpar09/tech/full_papers/ganapathi/ganapathi.pdf`.

[42] Robert Glück and Neil D. Jones. “Automatic Program Specialization by Partial Evaluation: An Introduction”. In: *Software Engineering in Scientific Computing*. 1996, pp. 70–77.

[43] Ekaterina Gonina. “Fast Speaker Diarization Using a Specialization Framework for Gaussian Mixture Model Training”. MA thesis. EECS Department, University of California, Berkeley, 2011. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-128.html`.

[44] Ekaterina Gonina. *PyCASP: Python-based Content Analysis Using Specialization*. 2012. URL: `http://www.eecs.berkeley.edu/~egonina/pycasp.html`.

[45] Martin Griebl, Christian Lengauer, and Sabine Wetzel. “Code Generation in the Polytope Model”. In: *In IEEE PACT*. IEEE Computer Society Press, 1998.

[46] *Groovy: A Dynamic Language for the Java Platform*. 2012. URL: `http://groovy.codehaus.org`.

[47] William Gropp. “MPICH2: A New Start for MPI Implementations”. In: *European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2002, pp. 7–. ISBN: 3-540-44296-0. URL: `http://dl.acm.org/citation.cfm?id=648139.749473`.

[48] David Heinemeier Hansson. *Active Record: Object-relation Mapping Put on Rails*. http://ar.rubyonrails.com/ 2004. URL: `http://ar.rubyonrails.com/`.

[49] A. Hellesoy and M. Wynne. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers. O’Reilly Vlg. GmbH & Company, 2012. ISBN: 9781934356807. URL: `http://books.google.com/books?id=oMswygAACAAJ`.

[50] M. D. Hill and A. J. Smith. “Evaluating Associativity in CPU Caches”. In: *IEEE Trans. Comput.* 38.12 (1989), pp. 1612–1630. ISSN: 0018-9340. DOI: `http://dx.doi.org/10.1109/12.40842`.

[51] Mark Frederick Hoemmen. “Communication-Avoiding Krylov Subspace Methods”. PhD thesis. EECS Department, University of California, Berkeley, 2010. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html`.

[52] Paul Hudak. “Building Domain-Specific Embedded Languages”. In: *ACM Comput. Surv.* 28 (4es 1996), p. 196. ISSN: 0360-0300. DOI: `http://doi.acm.org/10.1145/242224.242477`.

[53] Paul Hudak and Mark P. Jones. *Haskell vs. Ada vs. C++ vs. Awk vs...An Experiment in Software Prototyping Productivity*. Tech. rep. YALEU/DCS/RR-1049. New Haven, CT: Yale University Department of Computer Science, 1994.

[54] Intel. *Math Kernel Library*. URL: `http://developer.intel.com/software/products/mkl/`.

[55] *Intel Cilk Plus*. 2012. URL: `http://software.intel.com/en-us/intel-cilk-plus`.

[56] A. Jain. *pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures*. UC Berkeley EECS MS Report,see bebop.cs.berkeley.edu. 2008.

[57] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open Source Scientific Tools for Python*. 2001–. URL: `http://www.scipy.org/`.

[58] Neil D. Jones. "Transformation by Interpreter Specialisation". In: *Science of Computer Programming* 52 (1-3 2004), pp. 307–339. ISSN: 0167-6423.

[59] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. "An Auto-Tuning Framework for Parallel Multicore Stencil Computations." In: *IEEE International Parallel & Distributed Processing Symposium*. 2010, pp. 1–12.

[60] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. "Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations". In: *Workshop on Memory System Performance*. MSP '05. Chicago, Illinois: ACM, 2005, pp. 36–43. ISBN: 1-59593-147-3. DOI: `10.1145/1111583.1111589`. URL: `http://doi.acm.org/10.1145/1111583.1111589`.

[61] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. "Implicit and Explicit Optimizations for Stencil Computations". In: *Workshop on Memory System Performance and Correctness*. ACM Press, 2006, pp. 51–60.

[62] Jeremy V. Kepner and J. R. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011. ISBN: 9780898719901. URL: `http://www.worldcat.org/isbn/9780898719901`.

[63] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*. 2008. URL: `http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`.

[64] Ariel Kleiner, Ameet Talwalkar, Purnamrita Sarkar, and Michael I. Jordan. "A Scalable Bootstrap for Massive Data". In: (2012). URL: `http://arxiv.org/abs/1112.5016v2`.

[65] Andreas Klockner. *CodePy*. 2012. URL: `http://mathema.tician.de/software/codepy`.

[66] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. "PyCUDA: GPU Run-Time Code Generation for High-Performance Computing". In: *CoRR* abs/0911.3456 (2009).

[67] Andrew Koenig. "Patterns and Antipatterns". In: *The Patterns Handbooks*. Ed. by Linda Rising. New York, NY, USA: Cambridge University Press, 1998, pp. 383–389. ISBN: 0-521-64818-1. URL: `http://dl.acm.org/citation.cfm?id=301570.301985`.

[68] Christopher D. Krieger and Michelle Mills Strout. “Executing Optimized Irregular Applications Using Task Graphs Within Existing Parallel Models”. In: *Workshop on Irregular Applications: Architectures and Algorithms (IA^3) held in conjunction with SC12*. 2012.

[69] Chris Lattner. “LLVM and Clang: Next Generation Compiler Technology”. In: *Proceedings of BSDCan: The BSD Conference*. 2008.

[70] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, 2004.

[71] Daan Leijen and Erik Meijer. “Domain Specific Embedded Compilers”. In: *DSL*. Ed. by Thomas Ball. ACM, 1999, pp. 109–122. ISBN: 1-58113-255-7.

[72] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. “Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication”. In: *PKDD*. Springer, 2005, pp. 133–145. DOI: 10.1.1.111.8229.

[73] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. In: *Proceedings of the VLDB Endowment* 5.8 (Apr. 2012), pp. 716–727. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=2212351.2212354.

[74] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. “A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis”. In: *SDM’12*. 2012, pp. 930–941. URL: \href{http://siam.omnibooksonline.com/2012datamining/data/papers/158.pdf}.

[75] Grzegorz Malewicz, Matthew Austern, Aart Bik, James Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. “Pregel: A System For Large-Scale Graph Processing”. In: *SIGMOD* (2010).

[76] John D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[77] K.C. McPhee, C. Denk, Z. Al Rekabi, and A. Rauscher. “Bilateral Filtering of Magnetic Resonance Phase Images”. In: *Magnetic Resononance Imaging* (2011).

[78] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and How to Develop Domain-Specific Languages”. In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892. URL: http://doi.acm.org/10.1145/1118890.1118892.

[79] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. “Principles of Runtime Support for Parallel Processors”. In: *International Conference on Supercomputing*. ICS ’88. St. Malo, France: ACM, 1988, pp. 140–152. ISBN: 0-89791-272-1. DOI: 10.1145/55364.55378. URL: http://doi.acm.org/10.1145/55364.55378.

[80] Marghoob Mohiyuddin. “Tuning Hardware and Software for Multiprocessors”. PhD thesis. EECS Department, University of California, Berkeley, 2012. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-103.html.

[81] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Kathy Yelick. "Minimizing Communication in Sparse Matrix Solvers". In: *Supercomputing 2009*. Portland, OR, 2009.

[82] T. K. Moon. "The Expectation-Maximization Algorithm". In: *IEEE Signal Processing Magazine* 13.6 (Nov. 1996), pp. 47–60. ISSN: 10535888. DOI: 10.1109/79.543975. URL: http://dx.doi.org/10.1109/79.543975.

[83] G. E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (Apr. 1965), pp. 114–117. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[84] Jeffrey Morlan. "Auto-tuning the Matrix Powers Kernel with SEJITS". MA thesis. EECS Department, University of California, Berkeley, 2012. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-95.html.

[85] Chris J. Newburn et al. "Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language". In: *IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 224–235. ISBN: 978-1-61284-356-8. URL: http://dl.acm.org/citation.cfm?id=2190025.2190069.

[86] Eric Niebler. "Proto: A Compiler Construction Toolkit for DSELs". In: *Symposium on Library-Centric Software Design*. LCSD '07. Montreal, Canada: ACM, 2007, pp. 42–51. ISBN: 978-1-60558-086-9. DOI: 10.1145/1512762.1512767. URL: http://doi.acm.org/10.1145/1512762.1512767.

[87] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. 2007. URL: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[88] Travis E. Oliphant. "Python for Scientific Computing". In: *Computing in Science & Engineering* 9.3 (2007), pp. 10–20. URL: http://link.aip.org/link/?CSX/9/10/1.

[89] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. May 2008. URL: \url{http://www.openmp.org/mp-documents/spec30.pdf}.

[90] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, 1999. URL: http://ilpubs.stanford.edu:8090/422/.

[91] Michael Paleczny, Christopher Vick, and Cliff Click. "The Java HotSpot Server Compiler". In: *Java Virtual Machine Research and Technology Symposium (JVM '01)*. 2001.

[92] Jens Palsberg and C. Barry Jay. "The Essence of the Visitor Pattern". In: *International Computer Software and Applications Conference*. COMPSAC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 9–15. ISBN: 0-8186-8585-9. URL: http://dl.acm.org/citation.cfm?id=645980.674267.

[93] Inc. PLT Scheme. *The Racket Language*. 2012. URL: http://racket-lang.org.

[94] Sebastian Pop, Albert Cohen, Cdric Bastoul, Sylvain Girbal, Georges andr Silber, and Nicolas Vasilache. "GRAPHITE: Polyhedral Analyses and Optimizations for GCC". In: *GCC Developers Summit*. 2006, p. 2006.

[95] Aakash Prasad, David Howard, Shoaib Kamil, and Armando Fox. “Parallel High Performance Statistical Bootstrapping in Python”. In: *Scientific Computing in Python Conference*. 2012.

[96] L. Prechelt. “An Empirical Comparison of Seven Programming Languages”. In: *IEEE Computer* 33.10 (2000), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/2.876288.

[97] Calton Pu, Henry Massalin, and John Ioannidis. “The Synthesis Kernel”. In: *Computing Systems* 1 (1988), pp. 11–32.

[98] Markus Püschel et al. “SPIRAL: Code Generation for DSP Transforms”. In: *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”* 93.2 (2005), pp. 232–275.

[99] *PyPi - The Python Package Index*. 2012. URL: http://pypi.python.org/pypi.

[100] *Python Programming Language - Offical Website*. 2012. URL: http://www.python.org.

[101] Gabriel Rivera and Chau-Wen Tseng. “Tiling Optimizations for 3D Scientific Computations”. In: *ACM/IEEE conference on Supercomputing*. Supercomputing ’00. Dallas, Texas, United States: IEEE Computer Society, 2000. ISBN: 0-7803-9802-5. URL: http://dl.acm.org/citation.cfm?id=370049.370403.

[102] *Ruby Programming Language*. 2012. URL: http://www.ruby-lang.org.

[103] Armando Solar Lezama. *Program Synthesis By Sketching*. Tech. rep. UCB/EECS-2008-176. EECS Department, University of California, Berkeley, 2008. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-176.html.

[104] David Stutz, Ted Neward, and Geoff Shilling. *Shared Source Cli Essentials*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2002. ISBN: 059600351X.

[105] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. “The Pochoir Stencil Compiler”. In: *Symposium on Parallelism in Algorithms and Architectures*. SPAA ’11. San Jose, California, USA: ACM, 2011, pp. 117–128. ISBN: 978-1-4503-0743-7.

[106] David Tarditi, Sidd Puri, and Jose Oglesby. “Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2006, pp. 325–335.

[107] *Tcl Developer Site*. 2012. URL: http://www.tcl.tk.

[108] *The Perl Programming Language*. 2012. URL: http://www.perl.org.

[109] *The Programming Language Lua*. 2012. URL: http://www.lua.org.

[110] *The Scala Programming Language*. 2012. URL: http://www.scala-lang.org.

[111] Scott Thibault, Charles Consel, Julia L. Lawall, and Renaud Marlet Gilles Muller. “Static and Dynamic Program Compilation by Interpreter Specialization”. In: *Higher-Order and Symbolic Computation*. 2000, pp. 161–178.

[112] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. “A Scalable Auto-tuning Framework for Compiler Optimization”. In: *IEEE International Symposium on Parallel&Distributed Processing*. IPDPS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161054. URL: http://dx.doi.org/10.1109/IPDPS.2009.5161054.

[113] *Top500 Project: Top500 Supercomputing Sites*. 2012. URL: http://top500.org.

[114] Leslie G. Valiant. “A Bridging Model for Parallel Computation”. In: *Communications of the ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: http://doi.acm.org/10.1145/79173.79181.

[115] Richard Vuduc, James Demmel, and Katherine Yelick. “OSKI: A library of automatically tuned sparse matrix kernels”. In: *Journal of Physics Conference Series* 16.i (2005), pp. 521–530.

[116] Richard W. Vuduc. “Automatic Performance Tuning of Sparse Matrix Kernels”. PhD thesis. Berkeley, CA, USA: University of California, 2004. URL: http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf.

[117] Michael F. Wehner, Leonid Oliker, and John Shalf. “Towards Ultra-High Resolution Models of Climate and Weather”. In: *International Journal of High Performance Computing Applications* 22.2 (2008), pp. 149–165.

[118] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. “Automated Empirical Optimization of Software and the ATLAS Project”. In: *Parallel Computing* 27.1–2 (2001), pp. 3–35. URL: www.netlib.org/lapack/lawns/lawn147.ps.

[119] Samuel Williams, Andrew Waterman, and David A. Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures.” In: *Communications of the ACM* (2009), pp. 65–76.

[120] Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. “Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms”. In: *Interational Conference on Parallel and Distributed Computing Systems (IPDPS*. 2008.

[121] Samuel Webb Williams. “Auto-tuning Performance on Multicore Computers”. PhD thesis. EECS Department, University of California, Berkeley, 2008. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html.

[122] David Wonnacott. “Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations”. In: *Parallel and Distributed Processing Symposium, International* 0 (2000), p. 171. ISSN: 1530-2075. DOI: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2000.845979.

[123] Richard Xia, Tayfun Elmas, Shoaib Ashraf Kamil, Armando Fox, and Koushik Sen. *Multi-level Debugging for Multi-stage, Parallelizing Compilers*. Tech. rep. UCB/EECS-2012-227. EECS Department, University of California, Berkeley, 2012. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-227.html.

[124] Katherine A. Yelick et al. “Titanium: A High-performance Java Dialect”. In: *Concurrency - Practice and Experience* 10.11-13 (1998), pp. 825–836. URL: http://dblp.uni-trier.de/rec/bibtex/journals/concurrency/YelickSPMLKHGGCA98.

[125] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: `http://dl.acm.org/citation.cfm?id=1863103.1863113`.